IF/Prolog V5.3

User's Guide

Siemens AG Austria

Is there

anything you would like to tell us about this manual?
Please send us your comments.

Siemens AG Austria
PSE KB B3
Gudrunstrasse 11
A-1100 Vienna
Austria

Fax.: +43-1-1707 56992

email: prolog@siemens.at

The information in this document is subject to change and does not represent a commitment on the part of Siemens AG Austria. The software described in this document is furnished under a license agreement. The software may be used or copied only in accordance with the terms of the agreement.

UNIX is a registered trademark in the United States and other countries, licensed exclusively through X/Open Company Limited.

X/Open and the X device are trademarks of X/Open Company Ltd.

Copyright ©Siemens AG Austria, 1999. All rights reserved.
The reproduction, transmission, translation or exploitation of this document or its contents is not permitted without express written authority. Offenders will be liable for damages. Delivery subject to availability; right of technical modifications reserved.

Contents

Contents iii

1 Preface 1

Target group . . . 1

Summary of contents . . . 2

2 A first Prolog session 5

2.1 Calling and exiting Prolog . . . 5

2.2 A first Prolog query . . . 6

2.3 A short Prolog example . . . 9

2.3.1 Entering data with an editor . . . 11

2.3.2 Listing clauses . . . 13

2.3.3 Asking questions . . . 14

2.4 Modifying a database with an editor . . . 16

2.5 Extending the database in a Prolog dialog . . . 19

2.6 Modifying the database in a Prolog dialog . . . 23

2.7 Storing clauses in a file . . . 25

3 Database and procedure execution 27

3.1 Clauses . . . 27

3.1.1 Rules . . . 28

3.1.2 Facts . . . 29

3.2 Predicates . . . 30

3.2.1 Compiled predicates . . . 31

3.2.2 Built-in predicates . . . 32
3.3 Database . . . 34
3.4 Unification . . . 34
3.5 Procedure execution . . . 37
3.5.1 Queries and init directives . . . 38
3.5.2 Calling a goal . . . 39
3.5.3 Unification attempt . . . 42
3.5.4 Searching for the next clause . . . 42
3.5.5 Proving subgoals . . . 43
3.5.6 Proving a goal . . . 44
3.5.7 Failing to prove a goal . . . 45
3.5.8 Activating backtracking . . . 46
3.5.9 Redoing a goal . . . 47
3.5.10 Redoing a clause . . . 48
3.6 Procedural predicates . . . 50
3.6.1 The cut (!/0) . . . 51
3.6.2 Loops in Prolog . . . 56

4 The Prolog system **61**
4.1 Organization of the Prolog system . . . 61
4.1.1 Calling Prolog . . . 62
4.1.2 Exiting Prolog . . . 68
4.2 Querying information about the system . . . 69
4.2.1 Querying system size parameters . . . 69
4.2.2 Querying system properties . . . 70
4.2.3 Querying other information . . . 72
4.2.4 Querying memory management . . . 72
4.3 Environment variables . . . 74
4.4 Interfaces of the Prolog system . . . 76

5 The input/output interface **79**

5.1 Input/output in the Prolog top-level dialog . . . 79

5.2 Elementary input/output in Prolog . . . 80

5.2.1 Input/output using current streams . . . 82

5.2.2 Explicitly specifying streams for input/output . . . 83

5.2.3 Querying information about input/output streams . . . 85

5.3 Working with files . . . 87

5.3.1 Editing files with an editor . . . 87

5.3.2 Reading a file into the database . . . 90

5.3.3 Writing the database to a file . . . 91

5.4 Process management and communication . . . 91

5.4.1 Predicates for process management . . . 91

5.4.2 Predicates for process communication with signals . . . 92

5.4.3 Predicates for process communication using pipes . . . 92

5.4.4 Predicates for process communication using sockets . . . 95

5.4.5 Predicates for computer network information . . . 97

6 The Prolog grammar rules **101**

6.1 Context-free grammar . . . 101

6.2 Simple grammar rules . . . 103

6.2.1 Non-terminals in a production rule . . . 104

6.2.2 Terminals in a production rule . . . 105

6.2.3 Deriving a start symbol . . . 106

6.2.4 Syntactic analysis . . . 107

6.3 Additional arguments . . . 107

6.4 Explicit Prolog goals . . . 110

6.5 Syntax of the production rules . . . 112

7 The Prolog module concept **115**
7.1 Modules . . . 117
7.1.1 Directives . . . 117
7.1.2 Logical and physical files . . . 118
7.2 Module objects . . . 119
7.2.1 Predicates . . . 120
7.2.2 Metapredicates . . . 121
7.2.3 Global variables . . . 125
7.3 Module interface . . . 126
7.3.1 Exporting predicates . . . 127
7.3.2 Exporting metapredicates . . . 128
7.3.3 Reexporting all public predicates of a module . . . 130
7.3.4 Reexporting specific public predicates of a module . . . 131
7.3.5 Changes to the module interface . . . 132
7.4 The module body . . . 132
7.4.1 Importing all public predicates of a module . . . 133
7.4.2 Importing specific predicates of a module . . . 134
7.4.3 Declaring predicates as distributed . . . 136
7.4.4 Declaring predicates as private . . . 136
7.4.5 Declaring predicates as dynamic . . . 137
7.4.6 Declaring metapredicates . . . 138
7.4.7 Defining operators . . . 139
7.5 Compiling and loading files . . . 141
7.5.1 Compiling Prolog texts and modules . . . 141
7.5.2 Loading compiled files . . . 144
7.5.3 Removing a module from the database . . . 146

8 Contexts **147**
8.1 Context jumps . . . 148
8.2 Exceptions . . . 150
8.3 Signals . . . 154
8.3.1 Global control of signal handling . . . 156
8.3.2 Passive signal handling . . . 157
8.3.3 Active signal handling . . . 158

9 Debugging **161**

9.1 Debugger settings . . . 161

9.1.1 Activating/deactivating box ports . . . 162

9.1.2 Debug mode of a predicate . . . 163

9.1.3 Further parameters for the debugger . . . 163

9.2 The Tracer . . . 164

9.2.1 Predicates for the trace . . . 164

9.2.2 Start trace . . . 165

9.2.3 Trace operation . . . 165

9.2.4 Stop trace . . . 167

9.3 Interactive debugger . . . 167

9.3.1 Debugger operation . . . 168

9.3.2 Breakpoints . . . 170

9.3.3 Controlling debugger execution . . . 173

9.3.4 Branching in the proof tree . . . 173

9.4 Debugging with the alphanumeric user interface . . . 174

9.5 Debugging under the OSF/Motif user interface . . . 183

9.5.1 Controlling execution . . . 184

9.5.2 Setting ports . . . 185

9.5.3 Setting explicit breakpoints . . . 186

9.5.4 The Tracer . . . 189

9.5.5 Displaying ancestors . . . 190

9.5.6 Debugger log . . . 190

9.6 Summary tables of debugger commands . . . 191

10 The C interface **195**

10.1 Active C interface . . . 196

10.1.1 Linking simple C functions . . . 201

10.1.2 Complex C functions . . . 205

10.1.3 Built-in macros and functions for connecting complex C functions . . . 211

10.2 Passive C interface . . . 218

10.2.1 Overview of the built-in data types and functions . . . 219
10.2.2 Working with the passive C interface . . . 221
10.3 C device drivers . . . 228
10.3.1 t_device structure . . . 228
10.3.2 Control functions . . . 232
10.3.3 Functions for implementing device drivers . . . 234
10.4 Linking Prolog and C . . . 238

11 Graphical User Interface for MS–Windows 243
11.1 The IF/Prolog Graphical User Interface . . . 243
11.2 IF/Prolog Main Window Menu Bar . . . 244
11.2.1 File Menu . . . 245
11.2.2 Edit Menu . . . 246
11.2.3 Active File Menu . . . 246
11.2.4 View Menu . . . 246
11.2.5 Debug Menu . . . 247
11.2.6 Options Menu . . . 247
11.2.7 Help Menu . . . 248
11.3 Main Window Toolbars . . . 248
11.3.1 Main Toolbar . . . 249
11.3.2 Debug Main Toolbar . . . 250
11.3.3 Debug Information Toolbar . . . 251
11.4 Prolog Console with Scrollbars . . . 253
11.4.1 Console Key Sequences . . . 254
11.5 Status Bar . . . 255
11.6 Database Browser . . . 256
11.6.1 Database Browser Window . . . 257
11.6.2 Database Tree Pane . . . 257
11.6.3 Database Item Pane . . . 257
11.6.4 Menu . . . 258
11.7 Options Dialog . . . 258

11.7.1 General Options . . . 259
11.7.2 Prolog Flags Options . . . 261
11.7.3 Preprocessor Options . . . 264
11.7.4 Configuration Options . . . 266
11.7.5 Cross-Reference Options . . . 267
11.8 Debugger Support . . . 268
11.8.1 Debugger Toolbars . . . 269
11.8.2 Accelerator Keys . . . 269
11.8.3 Breakpoint Dialog . . . 270
11.8.4 Set Breakpoint Dialog . . . 271
11.9 Linking Custom IF/Prolog Applications . . . 273
11.9.1 Runtime User Interface . . . 273
11.9.2 Development User Interface . . . 274
11.9.3 Release and Debug configurations . . . 274
11.9.4 Special GUI startup parameters . . . 275
11.9.5 Demos . . . 276
11.10 Using the IF/Prolog Application Wizard . . . 276
11.10.1 How to finish IF/Prolog project creation . . . 279
11.10.2 How to create IF/Prolog Console Application (Prolog sources only) . . . 280
11.10.3 How to create IF/Prolog Console Application (Prolog and C/C++ sources) . . . 281
11.10.4 How to create IF/Prolog GUI Application (Prolog sources only) . . . 281
11.10.5 How to create IF/Prolog GUI Application (Prolog and C/C++ sources, dialog based) . . . 281
11.10.6 How to create IF/Prolog GUI Application (Prolog and C/C++ sources, MFC window based) . . . 281

A Preparing Prolog applications for marketing **283**

B ASCII table **287**

C Glossary **291**

Bibliography **295**

Index **296**

Chapter 1

Preface

The IF/Prolog system from Siemens AG Austria is an implementation of the ISO Prolog standard (ISO = International Standardization Organization). This standard was prepared by ISO Working Group 17, comprising representatives from various national standardization bodies.

IF/Prolog also contains interfaces and predicates which extend the language and ensure compatibility with earlier versions of the product.

The Standard dictates us to supply a strictly conforming mode, where the Prolog system only accepts and supplies conforming language predciates. To invoke this mode, see the section on invoking IF/Prolog in the User's Guide.

Target group

This *IF/Prolog User's Guide* is intended for anyone wishing to develop Prolog programs for a system running under the UNIX, Windows and OS/2 operating systems.

As the manuals for IF/Prolog are not tutorials, you should have some basic knowledge of the Prolog language. In addition, you should be familiar with the basics of the operating system and know how to use one of the editors installed on your system. If you wish to work with the C language interface, you should also have experience of the C programming language.

For a description of the elements of the Prolog language and the built-in predicates, refer to the *IF/Prolog Reference Manual* [1]. The interfaces of the Prolog system to the OSF/Motif window management system and to the Informix database system, including their associated built-in predicates, are dealt with in the manuals *IF/Prolog OSF/Motif Interface* [3] and *IF/Prolog Informix Interface* [4]. The *IF/Prolog Constraints Package* manual [5] explains how to work with constraints for variables in IF/Prolog. The *IF/Prolog Quick Reference* [6] contains all the built-in predicates, arranged into groups by function and accompanied by a brief description.

Summary of contents

The *IF/Prolog User's Guide* describes how to work with IF/Prolog.

The *IF/Prolog User's Guide* describes all the concepts on which the Prolog system is based. The *IF/Prolog Reference Manual* [1] is intended purely as a source of reference; in addition to the syntax elements it also contains a description of all the built-in predicates and C functions, and the debugger commands.

The second and third chapter in this manual are an introduction for absolute novices to Prolog. If you are already familiar with Prolog, feel free to skip these chapters.

The next chapters describe the organization of the Prolog system and the Prolog language. The description covers the concepts of the input/output interface and a driver model, along with the organization of the predicates in modules, and their contexts.
The interactive debugger, which you can operate with three different user interfaces, is based on the concept of the extended box model which is presented in chapter 3.5.
The last chapter explains both how to link C functions in Prolog and how to call Prolog in a C program, and also how to implement device drivers in C.

The following notational conventions are used throughout this manual:

xxx	Syntax definitions are enclosed within a frame.
`bind`	Prolog language elements, operating system commands and outputs from the system are printed in `teletype` font.
Name	*Italics* are used to represent variable parts in inputs and outputs where you should substitute them with your own values.
[ ]	Square brackets denote optional entries in the syntax notation; the brackets are not part of the Prolog text.
[ ]	Square brackets in bold type are elements of the Prolog list notation and are part of the Prolog text.
{ }	Braces denote alternatives in the syntax notation; the brackets are not part of the Prolog text.
|	A bar denotes alternatives in the syntax notation.
( )	Parentheses are required parts of the Prolog predicate notation and is part of the Prolog text.
...	Ellipsis indicate that the preceding syntax element may be repeated.
`atom/1`	Predicates are specified in the form *Name*/*Arity*.
`nl/0/1`	Several predicates with the same name and different arities are denoted in this form.
[1]	A number in square brackets indicates a reference to another manual or a textbook. The number identifies the publication in the Bibliography at the end of the manual.

The following pictograms are also used:

i for important advice and related information.

! for warnings.

Chapter 2

A first Prolog session

Is Prolog already installed on your system?
If not, please have the Prolog software installed.

The following examples are intended for users who have never worked with Prolog before.

2.1 Calling and exiting Prolog

Calling Prolog

When you see the shell prompt (e.g. '$'), type in the following:

```
$ prolog -debug
```

Terminate your input with the ↩ key.
After typing the name of the executable program (`prolog`) one may specify one or more options (e.g. `-debug`) (see also chapter 4, section entitled *Calling Prolog*). These options set the Prolog system in a specific initial state. The `-debug` option ensures that all user-defined predicates are implicitly declared as dynamic (see page 137), thereby enabling debugging of programs. The Prolog system is then loaded into main memory. When ready, it displays the welcome screen:

```
IF/Prolog <version> <operating system>
Copyright (C) Siemens Nixdorf Informationssysteme AG, 1996
Serial <serial> <license>

[user] ?-
```

'`[user] ?-` ' is the Prolog prompt. It indicates that you are at the Prolog top level and that Prolog is waiting for your input.

Exiting Prolog

You can now start experimenting with Prolog. But first you should know how to get out of Prolog so that you can exit whenever you need to:
You may make either a temporary exit or a permanent exit. When you exit Prolog temporarily, you can have commands executed in a subshell and then return to the Prolog system. This is particularly useful if you have already done some work with Prolog but have not yet saved your database. Quitting Prolog completely at such a stage would result in the loss of all your dynamic data.

To exit temporarily from Prolog, type:

```
[user] ?- system.↩
```

i You must terminate every input in a Prolog dialog with a period and the ↩ key.

The command interpreter prompt now appears on the screen, indicating that you can continue working in the operating system. As soon as you type `exit`, the Prolog system again issues its prompt:

```
$ exit↩

yes

[user] ?-
```

To quit Prolog completely, type:

```
[user] ?- halt.↩
```

You can also quit the Prolog system by pressing the key combination CTRL D. The shell prompt character will now appear. If you want to work with Prolog again at a later time, you will have to reinvoke Prolog with the command `prolog`.

2.2 A first Prolog query

Enter as your first query (or question):

```
[user] ?- write(hello).↩
```

The built-in predicate `write/1` generates the following output on your screen:

```
hello
yes

[user] ?-
```

i The number following a Prolog predicate (e.g. `1`) indicates how many arguments the predicate requires. This number is called the **arity** of the predicate.

Special cases

Character strings containing a blank, a period or some other special character must be enclosed in single quotes.

Example

If you want the character string `hello.` to be displayed with a terminating period, you must type the following:

```
[user] ?- write('hello.').↩
hello.
yes

[user] ?-
```

If you want `Hello` to be displayed with an uppercase H, you have to enclose it in single quotes because Prolog treats strings beginning with an uppercase letter as variables:

```
[user] ?- write('Hello').↩
Hello
yes

[user] ?-
```

i Any character string enclosed in single quotes is an atom.
Complete and detailed information about which character strings are regarded as atoms in Prolog can be found in the section *Prolog syntax* in the *IF/Prolog Reference Manual* [1].

Possible errors and how to correct them

1. Maybe you forgot to terminate your input with a period.
   If Prolog did not respond as described to `write(hello)`, then you may have forgotten to type the period '.' at the end of the line. If so, simply type the period and then hit the ↩ key. Your input sequence would then appear as follows:

   ```
   [user] ?- write(hello)↩
   | .↩
   hello
   yes

   [user] ?-
   ```

   The prompt character "`|`" signifies that input of the term has not yet been completed.

2. Maybe you wrote `Hello` with an uppercase H, but forgot to enclose it in single quotes. Prolog's response will be something like this:

   ```
   [user] ?- write(Hello).↩
   _656
   Hello   = _656
   ```

   If you now hit the ↩ key again, the Prolog prompt '`[user] ?- `' will appear once more:

   ```
   yes

   [user] ?-
   ```

   Hello is a Prolog variable because it begins with an uppercase H. The variable is not instantiated, i.e. no value has been assigned to it. For uninstantiated variables the system outputs an underscore character followed by a number. This is how variables are represented internally in the Prolog system.

   You must enclose the correct input in single quotes. It then appears as follows:

   ```
   [user] ?- write('Hello').↩
   Hello
   yes

   [user] ?-
   ```

3. Maybe you entered a period within the parentheses without enclosing the entire parenthesized expression in single quotes. Prolog regards the first period as the end of the query and responds with an error message such as:

   ```
   [user] ?- write(hello.).↩
   ```

```
*** syntax_error: operator expected
>>> line ?, column ?
```

In this example and the ones below the question mark stands for the line and the column number in which the error occurred in the sequence of the inputs. The correct input, with single quotes, is:

```
[user] ?- write('hello.').↩
hello.
yes

[user] ?-
```

4. The Prolog system could also respond with the following error message if, for example, you attempt to modify your input using the cursor keys:

```
*** syntax_error: illegal character
>>> line ?, column ?
[user] ?-
```

i You can correct input only in the current line by using the Backspace key

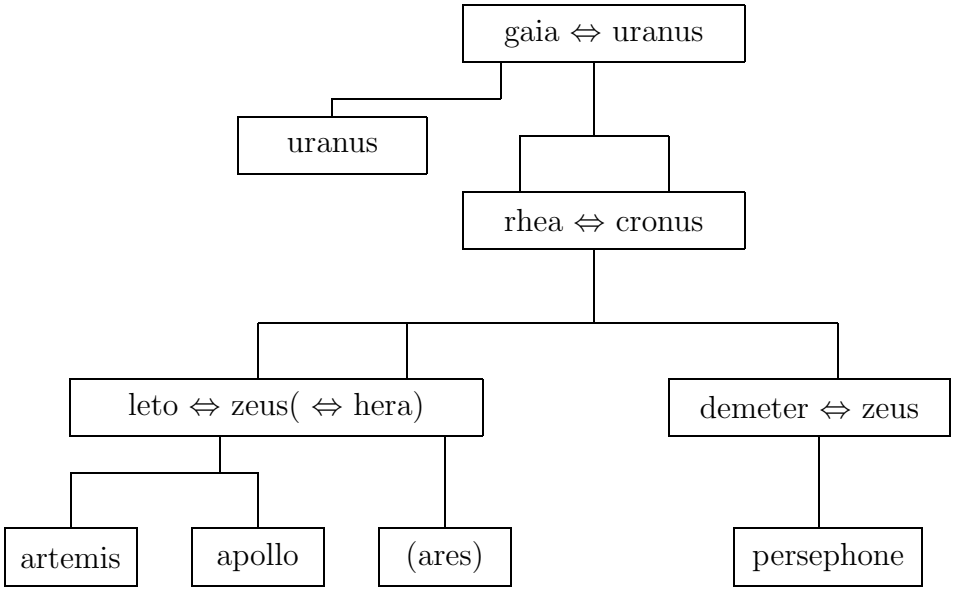
5. If all else fails, for example if you get no response with any of the usual keys, try hitting the Interrupt key (DEL key or the key combination CTRL C). This causes processing of the current query to be interrupted and a message to be displayed, followed by the Prolog dialog prompt '[user] ?- '.

```
[user] ?- repeat, fail.↩
DEL
*** E X C E P T I O N: system_error
>>> error = interrupt
>>> signal = interrupt
>>> goal = fail

no
[user] ?-
```

2.3 A short Prolog example

As the next step you should now write a small example database. This will contain facts and rules which describe the subject of the program.

The subject of the example is Greek mythological deities and the family relationships between various gods and goddesses.



⇔ signifies a relationship between a goddess and a god
— signifies a parent-child relationship

The relationships shown in the family tree are now described using Prolog.
The simplest relationship is a **fact**, for example: "Ares is male". In Prolog you can formulate the fact as follows:

```
male(ares).
```

The parent-child relationship or the siblings relationship requires a more detailed description, known as a **rule**. The rule for the siblings relationship could be formulated as follows: "X and Y are siblings if both have the same parents and if X and Y are not identical". Converted into Prolog, the rule looks like this:

```
sibling(X,Y) :-
        parentS(F,M,X),
        parentS(F,M,Y),
        X \= Y.
```

The comma between the individual subgoals indicates a conjunction. The rule is satisfied only when all three subgoals are true. The generic term for a fact or a rule in Prolog is a **clause**. For a detailed explanation of how to form syntactically correct Prolog clauses, refer to chapter 3.

In this first sample session with Prolog you will learn how to read valid Prolog clauses into the database and how to delete them from the database. In other words, you will know how to modify the database to suit your own requirements. This can be done with or without an editor:

- You can work with an editor by using the built-in predicate `edit/1`. Under UNIX, `edit/1` calls the **vi** editor by default and opens the file which you name. The following

sections refer specifically to the use of this editor.
If you want to use some other editor, e.g. **emacs**, then you can select it by calling the predicate

```
[user] ?- default_editor(_, emacs).↩
```

Your preferred editor must, of course, already be installed on your system. Any subsequent calls of `edit/0/1` will then execute this editor, as long as Prolog is active. You can use the environment variable `EDITOR` to preselect a different editor on operating system level.
When you exit the editor, the predicate `edit/0/1` returns you to Prolog level and, after asking for confirmation, reads your new or modified file into the database.

- You can modify the database directly on Prolog level without using an editor. The predicate `assertz/1/2` inserts an individual clause into the database, and the predicate `retract/1/2` deletes an individual clause from the database. This allows you to modify and test a program without switching between Prolog and the editor program and having to store the data in a file.

2.3.1 Entering data with an editor

After your first experiments with a Prolog query, you are still on the top level of Prolog and will see the Prolog prompt '`[user] ?- `' on your screen. If you have quit Prolog in the meantime, and do not get the prompt, call Prolog again. Using the built-in predicate `edit/1`, you can now open a file with the name *gods.pro* and type in a few facts and rules.

```
[user] ?- edit(gods).↩
```

You do not need to explicitly specify the suffix *.pro* for the file name, but you can if you wish. If you do not supply it, it is added automatically by the Prolog system. If you specify *.pro*, it must be enclosed in single quotes as it contains a period. If the file does not yet exist, it is created automatically in the current directory.
The selected editor is now called and you can enter the following facts and rules:

```
/* FACTS */
mother(gaia,uranus).
mother(X,Y) :- parentS(_,X,Y).

parentS(uranus,gaia,rhea).

parentS(cronus,rhea,zeus).
parentS(cronus,rhea,hera).
parentS(cronus,rhea,demeter).
```

```
parentS(zeus,leto,artemis).
parentS(zeus,leto,apollo).
parentS(demeter,zeus,persephone).

female(artemis).
female(persephone).
female(X) :- mother(X,_).

male(apollo).

father(X,Y) :- parentS(X,_,Y).
```

i In this example the functor `parentS` (=plural) is written with an uppercase "S" to distinguish it more clearly from the functor `parent` (=one parent).
Atoms, such as `gaia`, are written with a lowercase initial letter. Character strings which begin with an uppercase letter are **variables**.
Comments are introduced by '`/*`' and terminated by '`*/`'. End-of-line comments are introduced by the percent sign %. Comments have no effect on program execution.

The rules for each predicate should always follow one another in your file; otherwise, the Prolog system will issue warnings. If you do not write clauses for a predicate consecutively, you should inform the Prolog system by using the `discontiguous` directive.

Save the file, which was created as *gods.pro* in the current directory, and quit the editor. The Prolog system now asks whether the file is to be read into the database:
`Should the file gods.pro be reconsulted (y/n)?`
Answer yes by typing **y** and hitting the ↩ key. The Prolog system cannot access the information until the file has been reconsulted, i.e. read in.
If your *gods.pro* file includes clauses containing syntax errors, the Prolog system now reports these errors. The incorrect clauses are not read into the database.

The Prolog system confirms completion of the consult process:
`*** reconsult 'gods.pro': loaded in 0.02 sec.`
If your file contains syntactically invalid clauses, you will have to re-edit it (see page 16) to correct the errors.

Interpreting clauses

You can interpret the Prolog clauses you have entered as follows:

Prolog fact	Our interpretation
mother(gaia,uranus).	Gaia is the mother of Uranus.
parentS(uranus,gaia,rhea).	Uranus and Gaia are the parents of Rhea.
female(artemis).	Artemis is female.

Rules allow you to derive further information from the information already in the database. If you wish to represent a mother-child relationship using the existing information, there are two possibilities:

1. As the example family tree is so small, you could write all mother-child relationships represented in it into the database as facts.

2. You could deduce the mother-child relationship from the parent facts.

The second method is both neater and more efficient because your database will not always be so small. The following rule formulates the mother-child relationship:
`mother(X,Y) :- parentS(_,X,Y).`
This rule states that X is the mother of Y if there is a clause `parentS(_,`X`,`Y`)`. The first argument in the `parentS` clause (in this interpretation, the father) is of no significance for the rule. This is indicated by the inclusion of the anonymous variable `_` at this position.

Possible errors

If a clause in your *gods.pro* file contains a syntax error, the clause will not be read into the database. Instead, an appropriate syntax error message will be displayed on your screen.

In order to correct the error, you can

- modify the clause in your file with the predicate `edit/0/1`. The change will subsequently be stored and will be present in the database (see also page 19).

- modify the clause directly in the database with the predicate `assertz/1`. The clause will then be present in the database, but is not stored in the file. It will be lost when you quit Prolog.

2.3.2 Listing clauses

First, request an output of a list of the clauses which you have just read in from the *gods.pro* file on your screen. You do this by typing:

`[user] ?-` **`listing.`**↩

The predicate `listing/0` causes all rules and facts stored in the database to be output on the screen. In order to pause the output, press the key combination CTRL S. To resume, press CTRL Q. To stop the clauses being listed, press the DEL key.

If the predicate `listing/0` displays only `yes` on your screen, the reason may be that you have not declared your predicates as dynamic. Only dynamic predicates can be listed. By default Prolog regards all user-defined predicates as static. If you call Prolog with the `-debug` option, all user-defined predicates are implicitly declared as dynamic.

i If you are still debugging a Prolog program, you should always call the Prolog system with the `-debug` option.

2.3.3 Asking questions

You can now ask Prolog questions about the portion of the Greek gods' family tree represented in the database. Your first question (or query) is whether `gaia` is the mother of `uranus`. In Prolog you ask this question in the following form:

```
[user] ?- mother(gaia,uranus).↩
```

Prolog finds the fact in the database and answers:

```
yes

[user] ?-
```

However, if you ask whether `gaia` is the mother of `zeus`, Prolog will answer `no`:

```
[user] ?- mother(gaia,zeus).↩

no

[user] ?-
```

Prolog can do more than answer questions with yes or no. You can also ask a question such as "Which members of the divine family are female?":

```
[user] ?- female(Goddess).↩
```

Goddess is a variable because the string starts with an uppercase letter. The name you give to the variable is not important (you could, for example, equally well write `female(X).`) Prolog now searches the database for the first clause with the functor `female` and arity 1 in order to find an instantiation of the variable *Goddess*. The first matching clause in the database is the fact `female(artemis)`. You therefore receive the following answer to your question:

```
Goddess = artemis
```

You can now choose to continue the search for further names or end the search if you are satisfied with this answer.

1. If you want to see more names, type a semicolon ';' and press the ↩ key. The semicolon initiates backtracking.
   Prolog now searches for a further instantiation of *Goddess*. With the query `female(`*Goddess*`)`, some names will appear more than once. For example, `rhea` will be repeated three times because `rhea` occurs in three different `parentS` facts. (There are of course ways of avoiding such repetitions.) If no (more) instantiation is possible, Prolog responds with

   ```
   no

   [user] ?-
   ```

2. If you don't want to see any further instances, then simply hit the ↩ key:

   ```
   [user] ?- female(Goddess).↩

   Goddess = artemis ↩

   yes
   [user] ?-
   ```

Now ask a question about mothers and their children.

```
[user] ?- mother(Mother,Child).↩

Mother  = gaia
Child   = uranus
```

By typing '**;**' and hitting the ↩ key after each answer, you will be shown, one by one, all mother-child relationships in the database.

Trap

If you entered all the rules and facts exactly as given above, then a small error has crept in. This becomes apparent when you are given the answer:

`Mother = zeus`
The error can be traced to the fact `parentS(demeter,zeus,persephone)`. This should have been specified as `parentS(zeus,demeter,persephone)` so that the correct name will be found at the second argument position when the `mother` rule is applied.

2.4 Modifying a database with an editor

If you have already called Prolog and do not want to quit again completely, there are various ways of working temporarily with an editor. The simplest option is to use the predicate `edit/0/1`, which you have used already when creating the *gods.pro* file. After exiting from the editor and returning to Prolog level, confirm in answer to the question from the Prolog system that you want the file to be read in again.

When the file is read in, `reconsult/1` overwrites all clauses in the database which have the same functor and arity as clauses in the modified file. This ensures that there is only one occurrence of each clause in the database, even if you re-edit and read in the same file several times.

Type:

```
[user] ?- edit(gods).↩
```

This calls the preselected editor and reads in the *gods.pro* file which you have just stored. You can now amend the erroneous line to: `parentS(zeus,demeter,persephone)`. Add a few more rules to the file and, if you like, you can also include comment lines.
This example is included in the IF/Prolog distribution as file *$PROROOT/DEMOS/standard/gods.pro*.

```
/*
Database Greek Gods
*/

:- discontiguous mother/2, male/1, female/1.

/* Facts: mother(Mother, Child) */
mother(gaia,uranus).

/* Facts: parentS(Father, Mother, Child) */

/* First Generation */
parentS(uranus,gaia,rhea).
parentS(uranus,gaia,cronus).

/* Second Generation */
```

```
parentS(cronus,rhea,zeus).
parentS(cronus,rhea,hera).
parentS(cronus,rhea,demeter).

/* Third Generation */
parentS(zeus,leto,artemis).
parentS(zeus,leto,apollo).
parentS(zeus,demeter,persephone).

/* Facts: female(name) male(name) */

female(artemis).
female(persephone).

male(apollo).

/* X is mother of Y if X is the second parent of Y */

mother(X,Y):-
        parentS(_,X,Y).

/* X is father of Y if X is the first parent of Y */

father(X,Y):- parentS(X,_,Y).

/* X is female iff X is mother

female(X):-
        mother(X,_).

/* X is male iff X is father */

male(X):-
        father(X,_).

/* X and Y are siblings iff
        F and M are parentS of X and
        F and M are parentS of Y and
        X and Y are not identical */

sibling(X,Y):-
        parentS(F,M,X),
        parentS(F,M,Y),
        X \= Y.
```

```
/* X is sister of Y iff
        X is female and
        X and Y are siblings */

sister(X,Y):-
        female(X),
        sibling(X,Y).

/* X is brother of Y iff
        X is male and
        X and Y are siblings */

brother(X,Y):-
        male(X),
        sibling(X,Y).

/* X is parent of Y iff
        X is father of Y OR iff
        X is mother of Y */

parent(X,Y):-
        father(X,Y).

parent(X,Y):-
        mother(X,Y).

/* X is ancestor of Y iff
        X is parent of Y OR iff
        X is parent of Z and
        Z is ancestor of Y */

ancestor(X,Y):-
        parent(X,Y).

ancestor(X,Y):-
        parent(X,Z),
        ancestor(Z,Y).
```

Terminate the editor to store the changes in the *gods.pro* file.
Answer the Prolog system's reconsult question by confirming that the *gods.pro* file is to be read back into the database.

Special cases

You can delete individual clauses of a predicate from the database either by marking them as comment lines in your *gods.pro* file, i.e. by enclosing them in `'/*'` and `'*/'`, or by deleting them from the file. However, it is not possible to delete all clauses of a predicate, for example all `parentS` facts, from the database in this way. If you delete all clauses of a predicate from the file, even though nothing will be read in for this predicate, the old clauses will remain unchanged in the database.
If, on the other hand, at least one new or modified clause exists in the file, the change will be read in and all the old clauses will be deleted.

For how to delete all clauses of a predicate from the database, turn to page 23.

Possible errors

Clauses containing syntax errors will not be read into the database. A syntax error message will be displayed on the screen.

Asking questions

Now try asking questions about the information you have added to the database; for example:

```
[user] ?- ancestor(X,persephone).↩
```

Prolog outputs the ancestors of `persephone`. Again, the same names occur more than once, with those higher up the family tree appearing more frequently than those lower down. You could ask many more questions based on the patterns which you are now familiar with.

2.5 Extending the database in a Prolog dialog

To extend the database directly in a Prolog dialog, you use the built-in predicates `assertz/1` or `consult/1`.

Entering single clauses

To add a clause to the database, type the following when the Prolog prompt is displayed:

```
[user] ?- assertz(male(ares)).↩

yes

[user] ?-
```

`assertz/1` stores the specified clause into the database, placing it after all existing clauses with the same functor and the same arity.
In our example, the fact `male(ares)` is added after the existing `male` clauses.
There are other forms of the `assert` predicate, for example `asserta(`*Clause*`)`, which stores the clause as the first clause of the appropriate predicate. For a detailed description of all the available options, refer to the *IF/Prolog Reference Manual* [1].

If you make a syntax error when typing your input (omitting a parenthesis or period, for example), the clause containing the error is not written into the database. Prolog immediately issues an error message:

```
*** syntax_error: (type of error)
>>> line ?, column ?
```

You must repeat your last input with the clause specified correctly.

The `assert...` predicates store individual clauses into the database. If you want to enter a number of rules and facts, it is better to write these clauses directly into a file or to work in a Prolog dialog with `consult/1`.

Entering several clauses

Using the built-in predicate `consult/1` you can enter a number of clauses in succession in the database. This saves you having to call `assertz/1` separately for each clause. Type:

```
[user] ?- consult(user).↩
>
```

With `consult/1`, the atom `user` is used to address standard input and standard output. After this call, several clauses can be entered sequentially from standard input (usually the keyboard). Prolog indicates this by displaying a different prompt on standard output (usually the screen): `>`. You can now enter, line by line, all the facts and rules you need for your program. Remember to terminate each clause with a period, followed by the ↩ key. The Prolog system then immediately examines each clause for correct syntax. A syntactically correct clause is stored into the database immediately. Syntax errors are reported. You must then enter the erroneous clauses correctly.
Your final input is `end_of_file.` or the key combination CTRL D. The Prolog system then displays the prompt '`[user] ?- `' again.

If you take a closer look at the family tree for the Greek deities, you will come across some incestuous relationships. The following rules will enable such relationships to be deduced.

First, call `consult/1` and enter the following clauses in order to define these incestuous relationships:

```
[user] ?- consult(user).↩
```

```
> incest(X,Y):-
|      parentS(X,Y,_),
|      sibling(X,Y).
> incest(X,Y):-
|      parentS(X,Y,_),
|      father(X,Y).
> incest(X,Y):-
|      parentS(X,Y,_),
|      mother(Y,X).
> end_of_file.
> *** consult 'user': loaded in 0.03 sec.

yes
```

The prompt | means that the input of a term has not yet been completed. If you make a syntax error when typing your input after the > prompt, an error message will immediately appear on the screen. You should then re-enter the clause in the correct form.

Next, request a listing of your database:

```
[user] ?- listing.↩
```

The output produced by the predicate `listing/0` differs in respect of the order of the predicates and facts from that in your file:

```
% *** user: parent / 2 ***
parent(X,Y) :-
        father(X,Y) .
parent(X,Y) :-
        mother(X,Y) .

% *** user: ancestor / 2 ***
ancestor(X,Y) :-
        parent(X,Y) .
ancestor(X,Y) :-
        parent(X,Z),
ancestor(Z,Y) .

% *** user: incest / 2 ***
incest(X,Y) :-
        parentS(X,Y,_),
        sibling(X,Y) .
incest(X,Y) :-
        parentS(X,Y,_),
        father(X,Y) .
```

```
incest(X,Y) :-
        parentS(X,Y,_),
        mother(X,Y) .

% *** user: parentS / 3 ***
parentS(uranus,gaia,rhea).
parentS(uranus,gaia,cronus).
parentS(cronus,rhea,zeus).
parentS(cronus,rhea,hera).
parentS(cronus,rhea,demeter).
parentS(zeus,leto,artemis).
parentS(zeus,leto,apollo).
parentS(zeus,demeter,persephone).

% *** user: female / 1 ***
female(artemis).
female(persephone).
female(X) :-
        mother(X,_) .

% *** user: male / 1 ***
male(apollo).
male(X) :-
        father(X,_) .

% *** user: mother / 2 ***
mother(gaia,uranus).
mother(X,Y) :-
        parentS(_,X,Y) .

% *** user: father / 2 ***
father(X,Y) :-
        parentS(X,_,Y) .

% *** user: sibling / 2 ***
sibling(X,Y) :-
        parentS(A,B,X),
        parentS(A,B,Y),
        x \= Y .

% *** user: sister / 2 ***
sister(X,Y) :-
        female(X),
        sibling(X,Y) .
```

```
% *** user: brother / 2 ***
brother(X,Y) :-
        male(X),
        sibling(X,Y) .

yes
```

You can now ask a question, for example:

```
[user] ?- incest(zeus,Goddess).↩
```

or search for all incestuous relationships:

```
[user] ?- incest(X,Y).↩
```

Note the positions of the atom `zeus` and the variable *Goddess* in the question `incest(zeus,` *Goddess*`)`. As the rules for `incest` apply directly to the `parentS` facts, the sequence `incest(`*Father*`,` *Mother*`)` must be observed. If you were to ask `incest(`*Goddess*`, zeus)`, Prolog would answer `no`.

2.6 Modifying the database in a Prolog dialog

Prolog also provides built-in predicates for deleting clauses and predicates.

Deleting single clauses

To delete a clause from the database, you use the predicate `retract/1`:

```
[user] ?- retract(mother(gaia,uranus)).↩

yes
```

Now use the predicate `listing/0` to obtain a listing of the database: you will see that the fact has been deleted.
To insert the fact in the database again, enter:

```
[user] ?- assertz(mother(gaia,uranus)).↩

yes
```

Deleting all clauses of a predicate

To delete all clauses of a predicate from the database, you use the built-in predicate `abolish/1`. The following `abolish/1` call deletes all clauses of the predicate with functor `ancestor` and arity 2:

```
[user] ?- abolish(ancestor/2).↩

yes
```

The `ancestor` rules are now deleted from the database, but still exist in the *gods.pro* file.

i Prolog answers `yes` even if the clause you want to delete cannot be found in the database. You should therefore be careful when deleting clauses. If, for example, you mistype one of your argument entries, it could occur that the intended predicate is not deleted because it could not be found. In spite of this, no error message will be issued.

Modifying several clauses

If you want to modify a number of clauses in the database, the most convenient method is to use `edit/0/1`. This ensures that the changes are applied both to the file and to the database at the same time.
But if you prefer, you can also modify the database directly in a Prolog dialog:

```
[user] ?- reconsult(user).↩
>
```

or in abbreviated form:

```
[user] ?- [-user].↩
>
```

As in the case of `consult(user)`, you can now write all the clauses which you want to include in the new database. Your final input is `end_of_file.` or the key combination CTRL D.
The Prolog system then displays the prompt '`[user] ?- `' again.

i All old entries with the same functor and the same arity as the new. clauses will be overwritten in the database, and thus deleted. In other words, the database is modified by `reconsult`, rather than merely extended as in the case of `assertz` or `consult`

As there are now no further errors in the example database, you can immediately terminate the modification process by typing `end_of_file.` or pressing the key combination CTRL D.

2.7 Storing clauses in a file

So far, all the clauses you have entered or modified in the Prolog dialog exist only in the database. If you want to quit Prolog without losing your facts and rules, then you will first have to store them in a file. The file name should be a new one, so that no existing file will be overwritten. Type the following:

```
[user] ?- tell('dei.pro'), listing, told.↩

yes
```

! If you name an existing file in the `tell/1` predicate, it will be overwritten.

Here, we have a query consisting of three subgoals. These are processed sequentially from left to right.
`tell/1` redefines the current output stream. Subsequent outputs will now be written to the specified file.
This is why the output from the `listing/0` predicate does not appear on the screen, but is written directly to the file *dei.pro*.
The predicate `told/0` closes the current output file again so that any further output will be directed to the screen.

The suffix `.pro` is in this case not automatically added to the file name. However, you should identify all files which contain Prolog clauses with this suffix.

Chapter 3

Database and procedure execution

This chapter opens with a brief overview of the elements and structure of a Prolog database and continues with a step-by-step description of the principles underlying procedure execution in Prolog. The basic features of an extended box model for Prolog predicates are also introduced and explained. This extended box model forms the basis of the Prolog debugger, described in greater detail in the chapter 9 on *Debugging* in this manual.

The following description is not intended as a tutorial on the Prolog language but because of the descriptive nature of Prolog, even a Prolog novice will be able to understand the simpler examples and thereby the underlying syntax and methodology of the Prolog language.

For a sample session illustrating the various ways of creating a database and explaining the predicates used for database manipulation, see chapter 2.

3.1 Clauses

In Prolog, a clause represents a statement about objects and relations between objects. Clauses represent the knowledge which the Prolog system can draw upon when processing a query.

The precise syntax of clauses is defined in the section on *Syntax* in the *IF/Prolog Reference Manual* [1].

If clauses are to be read in from a file or from the keyboard using `consult/1` or `reconsult/1`, each clause must end with a period and be separated from the next clause by a delimiter. This period does not itself form part of the clause syntax, but serves the Prolog system as an additional delimiter during syntactical analysis. When the clauses in the database are output with `listing/0` or `listing/1`, a period is again added to the end of each clause. This improves readability and has the added advantage that clauses output to a file in this way can be read back in again without the need for further manipulations (see also the section 5.3.3: *Writing the database to a file*).

There are two kinds of clauses: rules and facts.

3.1.1 Rules

A rule is a Prolog clause containing a clause body.

A rule represents conditional knowledge, i.e. it is an implication in the form "clause head <== clause body" (the clause head is true if the clause body is true). At the same time the clause head can be seen as the activation point of a procedure which activates further procedures in the clause body.

The most important operations between subgoals (conditions) in the clause body are the following:

operations	**notation**
conjunction (AND, comma)	`,`
disjunction (OR, semicolon)	`;`
IF-THEN (arrow)	`->`

As an alternative to operator notation using ',', ';' or '->', it is also possible (but rather inconvenient) to specify the clause body in the equivalent normal structure.

When specified explicitly, parentheses effect the precedence and hence the evaluation of the subgoals involved.

The clause body is true if

- for ANDed (conjoined) conditions (comma), all conditions have the truth value `TRUE`;
- for ORed (disjoined) conditions (semicolon), at least one of the conditions has the truth value `TRUE`.
- for IF-THEN conditions (arrow), both conditions have the truth value `TRUE`.
- If ANDs, ORs and IF-THENs are mixed, the whole evaluation is governed by parentheses and operator precedences. The principal functor of the clause body will be determined; then both the associated arguments are evaluated and finally, the truth value of the whole expression is determined on the basis of one of the three rules described above.

  The exception to this is a combination of an OR operation and an IF-THEN operation in the lefthand part of the OR operation. This special combination forms an IF-THEN-ELSE operation on conditions. This operation is true if either the IF condition and the THEN condition have the truth value `TRUE`, or the IF condition has the truth value `FALSE` and the ELSE condition has the truth value `TRUE`.

For reasons of clarity you should avoid using disjunctions in clauses and instead write two clauses; clauses with the same clause head have an implicit OR-relation.

```
clause_head :-
        subgoal1 ;
        subgoal2.
```

is equivalent to

```
clause_head :- subgoal1.

clause_head :- subgoal2.
```

Example

```
sibling(X,Y) :-              % The sibling rule has a clause body
        parentS(F,M,X),      % consisting of three subgoals joined
        parentS(F,M,Y),      % by AND.
        X \= Y.

half_sibling(X,Y) :-         % The half_sibling rule has a clause body
        ( parentS(F1,M,X),   % consisting of two conditions joined by
          parentS(F2,M,Y),   % OR.
          F1 \= F2,
          X \= Y
        )
        ;                    % The two conditions each have the form
        ( parentS(F,M1,X),   % "(clause body)", with each clause body
          parentS(F,M2,Y),   % consisting of four conditions joined by
          M1 \= M2,          % AND.
          X \= Y
        ).
```

It would be better to reformulate this rule as:

```
half_sibling(X,Y) :-
        parentS(F1,M,X),
        parentS(F2,M,Y),
        F1 \= F2,
        X \= Y .

half_sibling(X,Y) :-
        parentS(F,M1,X),
        parentS(F,M2,Y),
        M1 \= M2,
        X \= Y .
```

3.1.2 Facts

A fact is a Prolog clause without a clause body.

A fact represents unconditional knowledge, i.e. it describes a relation between objects which always holds true, regardless of any other relations. Thus, provided their clause heads are identical, a fact is equivalent to a rule whose clause body always has the truth value `TRUE`.

Example

```
parentS(paul, eve, john).              % john has parents eve and paul
female(anita).                         % anita is female
faculty(0,1).                          % 0! is 1
anita likes troll.                     % likes defined as infix operator
father_son(father(paul), son(john) ). % paul is the father of john
append([],L,L).                        % list concatenation: appending
                                       % any list L to the empty list
                                       % yields the list L
buddha_nature(_).                      % everything has Buddha nature
```

3.2 Predicates

Clauses with the same clause name, the same number of arguments and defined in the same module are combined in the database and form the definition of a **predicate**.

The common clause name is called the predicate name or **functor** of the predicate. The number of arguments is the **arity**. For example, the predicate `fac/2` is defined by the collection of all clauses with the clause head `fac(`*Arg1,Arg2*`)`, where *Arg1* and *Arg2* may be any terms.

The separate clauses of a predicate are connected by disjunction, i.e. by inclusive OR.

Clauses with the same clause name but a different number of arguments belong to different predicates. Likewise, clauses which have the same clause name and the same arity but are associated with different modules belong to different predicates.

The Prolog predicate concept can be compared to the subprogram concept in conventional programming languages; we therefore also speak of "calling" a predicate in Prolog. The predicate concept ensures a high degree of modularity in Prolog programs and thereby supports structured programming.

Example

The two clauses below define the predicate `fac/2` for calculating the factorial of a natural number.

```
fac( 0, 1) :- !.              % 0! is 1
```

```
fac( N, Nfac) :-
          N > 0,              % for integers N > 0
          M is N - 1,         % defines NFac as N * (N-1)!
          fac( M, Mfac),      % thus MFac is calculated
          Nfac is N * Mfac.   % recursively as (N-1)!
```

You can call the predicate `fac/2` in the Prolog input loop by entering, for example,

```
[user] ?- fac(5,X).↵
```

The Prolog system searches the clauses belonging to the predicate `fac/2` and displays the answer:

```
X = 120
```

The precise sequence in which Prolog performs this search operation is described in the section *Procedure execution*.

Only predicates whose clauses have been included in the database with `consult/1`, `reconsult/1` or with an `assert` predicate can be modified, i.e. clauses can be added (`assert` predicates) or removed (`retract` predicates), individual predicates can be deleted (`abolish/1`) or replaced (`reconsult/1`). The clauses of these predicates can be output with `listing/0/1` and analyzed with `clause` predicates. Note however that to do so you must call the Prolog system with the `-debug` option.

3.2.1 Compiled predicates

IF/Prolog offers a number of ways of compiling user-defined predicates and including them in the database. This can be done by the system's incremental compiler by using the `assert` and `consult` predicates. In addition, predicate definitions stored in files can be compiled by the predicate `compile/1` or the external compiler (`procmp` command).

Predicates compiled with `consult/1`, `reconsult/1`, `compile/1` or `procmp` are normally no longer modifiable. However, you can inform the compiler with the `dynamic/1` directive that certain predicates are to remain modifiable. Predicates which you have generated with `assert` predicates or which you have declared as modifiable with other compilation methods are compiled only to the extent where the compilation is reversible, i.e. where decompilation is possible.

Whenever references are made in this manual to **compiled predicates** they should be taken to mean predicates which cannot be decompiled. These can no longer be modified by simply adding or removing clauses. Further information about compiling Prolog source text and the related advantages and restrictions can be found in chapter 7.5 on *Compiling and loading files*.

C interface

IF/Prolog also gives you the option of using C functions without backtracking (simple C functions) and with backtracking (complex C functions) as Prolog predicates using the built-in C interface. The C interface is described in detail in chapter 10.

3.2.2 Built-in predicates

IF/Prolog provides a large number of built-in predicates which you can use to formulate your own predicates.

Built-in predicates cannot be extended by additional clauses or entirely replaced. The module system does, however, give you the option to define your own predicates with the same clause names and the same arity as built-in predicates, since all the built-in predicates are defined in special system modules.

An overview and an alphabetically arranged description of the built-in predicates can be found in the *IF/Prolog Reference Manual* [1].

Predicates which permit backtracking are indicated by a hash character ('`#`') in front of the functor. This character does not appear with predicates not having this capability.

For **metapredicates**, the argument list is followed by the text `[` @ `+`*Module* `]`. The module qualification can be specified by `@/2`. **Metapredicates** are supplied with information on the calling module.

Directives are special syntactic structures which can be specified in IF/Prolog texts and which are processed when these texts are read in (e.g. with `consult/1`). They are indicated by `:-` in front of the functor, as they would be written in a Prolog text.

User definable predicates are automatically called at particular points by IF/Prolog if the user has defined them. They are identified by the text `[ :-` *Body* `]` after the argument list.

The **call mode** specifies the instantiation of an argument at the time of the call. In front of each argument there is a sign ('`@`', '`+`', '`-`', or '`?`') to indicate the call type of the argument, as follows:

`@` The argument is a pure **input** parameter. The current parameter specified in the call must be of the prescribed type and any uninstantiated variables contained in this parameter are not instantiated in the call.

`+` The argument is an **input** parameter. The current parameter specified in the call must be of the prescribed type. Any uninstantiated variables contained in this parameter may be instantiated in the call.

When the argument is an atomic term, there is no difference between the modes `+` and `@`. The mode `@` is therefore used only when the argument may be a compound term.

? The argument is an **input/output** parameter. The current parameter must be either a variable or a term of the prescribed type. In the course of the execution of the predicate, this parameter is unified. If this unification is not successful, then the entire predicate call will fail. Any uninstantiated variables contained in this parameter may be instantiated in the call.

- The argument is a pure **output** parameter. The current parameter must be an uninstantiated variable. If the predicate succeeds, this variable is instantiated with the result of the predicate call. The type of result from the predicate call is indicated in the section on '*Arguments*' in the full description.

Several call patterns are possible for some predicates.

In some predicates where an integer is expected as an input parameter, it is also possible to input an arithmetic expression with an integer value; an expression will be evaluated implicitly as with `is/2`.

Examples

If uninstantiated variables are permitted in the mode `@`, they will not be instantiated.

The argument of `atom/1` has the call mode `@`. It is therefore possible to specify a variable in order, for example, to test whether it is instantiated to an atom. This variable will not be instantiated, however.

The three arguments of `current_op/3` have the call mode `?`. If all three arguments are uninstantiated variables, you can obtain all defined operators by means of backtracking:

```
[user] ?- current_op(Precedence,Type,Name).↩


Precedence = 50
Type       = xfx
Name       = : ;

Precedence = 100
Type       = xfx
Name       = @ ;    etc. until you terminate backtracking.
```

The instantiated arguments are unified; the uninstantiated variables are instantiated. The following queries operators with precedence 400:

```
[user] ?- current_op(400,Type,Name).↩
```

```
Type    = yfx
Name    = >> ;

Type    = yfx
Name    = <<

yes
```

3.3 Database

The database is the set of predicates available at the moment a goal is being processed. It includes:

- User-defined predicates
  - Predicates which the user defines by means of clauses. The relevant clauses for a predicate are those which were read in at system startup with `consult/1`, `reconsult/1` or an `assert` predicate but were not subsequently removed.
  - Compiled predicates loaded into the database by the user.
  - Predicates defined by the user with the C interface.
- Built-in predicates
  - The built-in predicates are made available automatically.

Note, however, that the contents of the database are not entirely visible. Visibility is controlled by the module concept (see section 7: *The Prolog module concept*).

3.4 Unification

The key concepts governing procedure execution are **unification** and **depth-first search**.

Unification is an operation which

- checks whether two terms are equal or can be made equal by suitably instantiating their uninstantiated variables,
- if necessary, performs an instantiation of the most general kind possible (substitution).

Otherwise unification fails. The unification operator (the equal sign) is declared with `op(700, xfx, =)` as a non-associative infix operator (see `=/2`).

Unification is governed by the following rules:

- an (uninstantiated) variable can be unified with any term by being instantiated to this term;
- constants are unifiable if they are identical;
- structures are unifiable if they have the same functor and the same arity and if corresponding arguments are also unifiable.

When a variable is instantiated to a term by unification, the instantiation applies to the particular clause involved. It is not possible to perform a new instantiation for this variable within the clause, and any subsequent unification with a different term will fail. Variable instantiation can only be undone in the course of backtracking.

The system does not check whether a variable to be replaced occurs a second time in the substitution term. This "occurs check", as it is called, is not performed because it takes a great deal of computer time. Consequently, when formulating your clauses you should ensure that loops of this sort (e.g. `X = f(X)`) do not occur. Unification with an occurs check is implemented by the built-in predicate `unify_with_occurs_check/2`.

Example

The Prolog program below tests whether the two terms *Term1* and *Term2* are unifiable; where applicable, it also defines a list of necessary substitutions as the unifier.

```
unification(Term1,Term2,Substitution) :- % The original term pair
  Stack = pair(Term1,Term2),              % is put in the stack
  unify([Stack],Substitution).            % Start unification

unify([],[]).                           % End if stack is empty

unify([pair(T1,T2)|R],Unificator) :-    % Take first pair from stack
  uni(T1,T2,R,R1,Sub1),                 % Find first substitutions Sub1
                                        % and create new stack R1
  unify(R1,Sub2),                       % Find remaining substitutions
  append(Sub1,Sub2,Unificator).         % there and combine them all to
                                        % form unifier

uni(T1,T2,R,R,[]) :-                    % Constants are unifiable
  atomic(T1),atomic(T2),!,T1 == T2.     % if they are identical.
                                        % No changes made to stack

uni(T1,T2,R,R,[]) :-                    % Identical terms can be unified
  T1 == T2,!.                           % without further substitution

uni(T1,T2,R,R1,[T1 = T2]) :-            % T1 is a variable; replace T1
  var(T1),stack_replace(T1,T2,R,R1).    % with T2 in stack and record the
```

```
                                        % substitution T1 = T2

uni(T1,T2,R,R1,[T2 = T1]) :-            % T2 is a variable; replace T2
  var(T2),stack_replace(T2,T1,R,R1).    % with T1 in stack and record the
                                        % substitution T2 = T1
uni(T1,T2,R,R1,[]) :-
  compound(T1),compound(T2),            % T1 and T2 are structures with
  functor(T1,Functor,Arity),            % the same functor and the same
  functor(T2,Functor,Arity),            % arity; push all argument
  push(T1,T2,Arity,R,R1).               % pairs onto the stack

push(_,_,0,R,R) :- !.                   % End because there are no more
                                        % arguments

push(T1,T2,N,R,[pair(X,Y)|R1]) :-       % Take the two Nth arguments,
  arg(N,T1,X),arg(N,T2,Y),              % put the pair on the top of the
  N1 is N - 1,                          % stack, push down the remaining
  push(T1,T2,N1,R,R1).                  % pairs beneath it.

stack_replace(_T1,_T2,[],[]) :- !.      % End because there is nothing
                                        % more to replace

stack_replace(T1,T2,[ pair(X_old,Y_old) | RR ],
    [pair(X_new,Y_new)|RR_new]) :-
  replace(T1,T2,X_old,X_new),           % Perform substitutions in the
  replace(T1,T2,Y_old,Y_new),           % first pair, then in the rest
  stack_replace(T1,T2,RR,RR_new).       % of the stack

replace(Old,New,Term,New) :-            % If Term is identical to Old,
  Term == Old.                          % replace it with New

replace(Old,_New,Term,Term) :-          % If Term is not a structure
  not compound(Term),!, Term \== Old.   % nor identical to Old, there
                                        % is nothing to replace

replace(Old,New,Term,Term1) :-          % If Term is a structure,
  functor(Term,Functor,Arity),          % form a second Term1 with
  functor(Term1,Functor,Arity),         % the same functor and arity
  str_rep(Arity,Old,New,Term,Term1).    % and substitute New for Old
                                        % in the arguments there

str_rep(0,_Old,_New,_Term,_Term1) :- !. % End because there are no more
                                        % arguments

str_rep(N,Old,New,Term,Term1) :-
```

```
  arg(N,Term,Arg),                    % Take the Nth argument from
  replace(Old,New,Arg,Arg1),          % Term and substitute New for Old.
  arg(N,Term1,Arg1),                  % The result Arg1 is the new Nth
  N1 is N - 1,                        % argument of Term1
  str_rep(N1,Old,New,Term,Term1).     % Do same for remaining arguments
```

```
[user] ?- unification(a(Y,b([])),a(YY,C),U).↩

Y       = _1288
YY      = _1312
C       = _1316
U       = [ _1316 = b([]),_1288 = _1312]

yes
```

The system displays the substitution which causes the terms to be equal. The built-in unification mechanism of Prolog is addressed by means of the predicate `=/2` :

```
[user] ?- f(a,b) = f(X).↩

no
```

The two terms are not unifiable because their arities do not match.

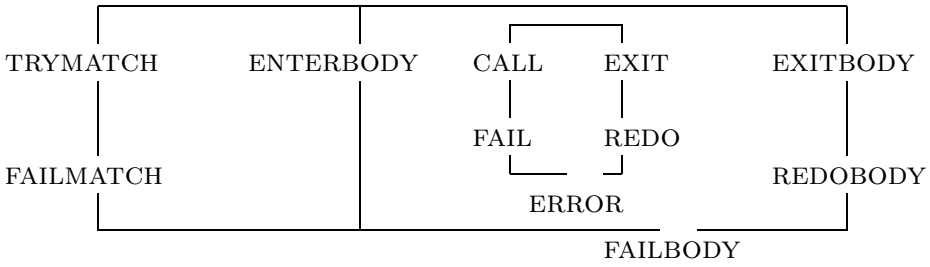
```
[user] ?- X = a, Y = [X].↩

X       = a
Y       = [a]

yes
```

3.5 Procedure execution

Effective programming in Prolog requires not only that the problem has been logically correctly structured, but also that you take into account the procedure execution defined by the Prolog search technique. This is essential whenever predicates with a clearly procedural character are used.

To allow you to maintain an overview of the actual state of goal execution even during backtracking or when unpredicted results occur, it is helpful to illustrate the situation by means of a model. Below, we introduce an extended box model as one means of providing

an especially detailed simulation of goal execution. This model contains, in addition to the ports `call, exit, redo` and `fail` of the "standard" box model, nine additional ports as a refinement, thereby permitting a more accurate modeling of goal execution. This model is referred to below as the **box model**.



Box Model

The execution of goal can be simulated on this box model as a path through the ports of boxes, during which further boxes can be created and, if need be, subsequently destroyed. The steps taken from port to port and the creation and destruction of boxes are described in detail in the following. Goal execution is divided into individual steps. Related steps have been combined in the description and are represented first in the order in which they are passed when a goal is being proven. In the course of backtracking, we return to boxes which were created but have not been destroyed again. These are precisely those boxes in which **choice points** are located.

Taking into account the individual circumstances involved, the entire procedure execution can then be reconstructed from small steps of this sort. The Prolog system debugger is derived from this model.

3.5.1 Queries and init directives

Program execution in conventional programming languages such as C or FORTRAN corresponds in Prolog to the execution of a goal, using the predicates of the database.

There are three different ways to activate the execution of a goal:

- Query
  Every correct input following the prompt '`[user] ?-` ' in the Prolog input loop is understood by Prolog to be a goal to be proved in the current database. The specified goal term should end with a period (.).

- `initialization` or `init` directive
  If the Prolog system consults or loads a file which contains one or more `initialization` or `init` directives, the goal terms they contain are proved after the load operation. The predicates defined in the relevant file are all available for proof purposes. Note that execution always takes place after the file has been loaded, irrespective of the position of the initialization directives in the original file.

- `program/0`
  When the Prolog system is started, a check is carried out after the command line arguments have been processed (i.e. in particular after the files specified in the command line have been loaded or consulted) to see whether there is a definition of the predicate `program/0` in the database in the module `user`. If so, instead of starting the normal Prolog input loop (`break/0`) the goal `user:program` is executed. After this operation the Prolog system terminates. Thus, it is possible to implement independent applications.

i Output of variable instantiations, interactive backtracking capability, and output of the proof result in the form `yes` or `no` is a property of the Prolog input loop (`break/0`) and therefore takes place only in the event of queries.

3.5.2 Calling a goal

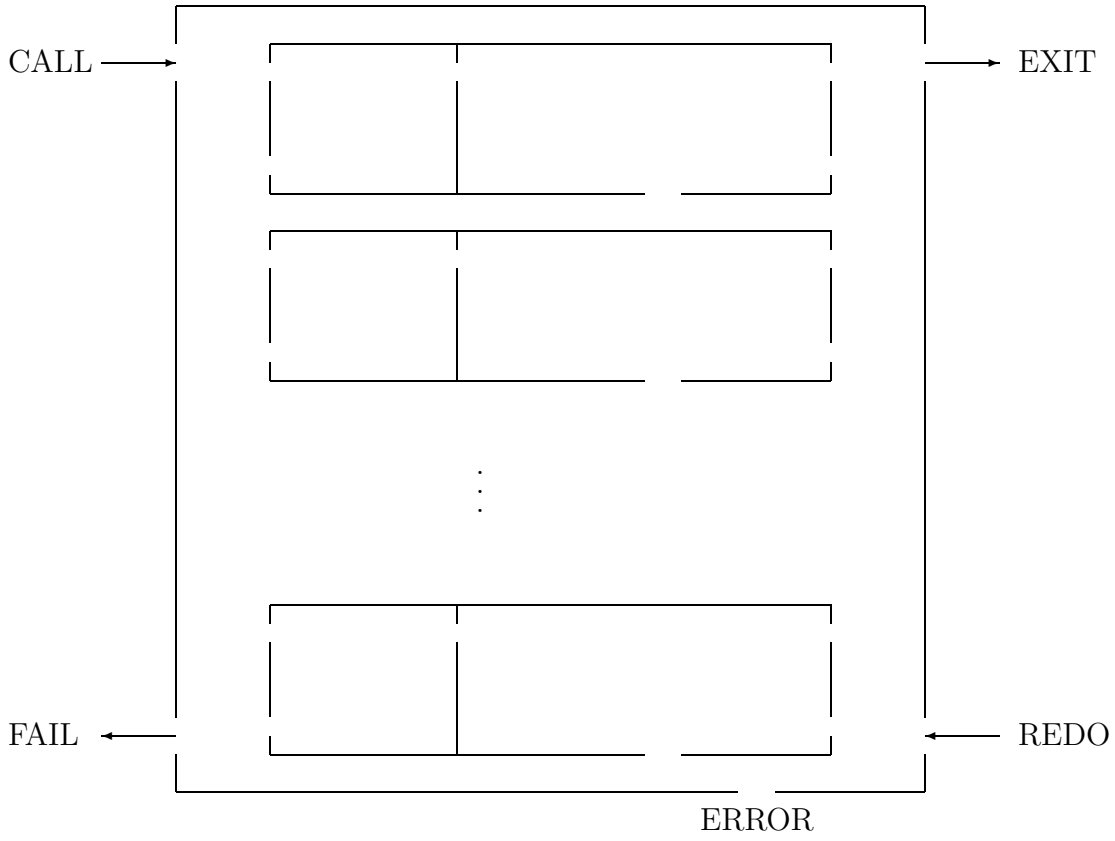
When you call a goal with several subgoals ($Subgoal_1, \ldots, Subgoal_n$), the Prolog system first tries to prove the first subgoal. To do this, it searches the database for the first clause of the matching predicate, i.e. the first clause which has the same clause name and arity as the term *Subgoal_1*. If more than one clause belongs to this predicate, the system will set a choice point through which it may, if necessary, reach the next clause of the predicate when backtracking.

A goal or subgoal is represented by a goal box with the five outermost ports `call, exit, redo, fail` and `error`.

Goal box

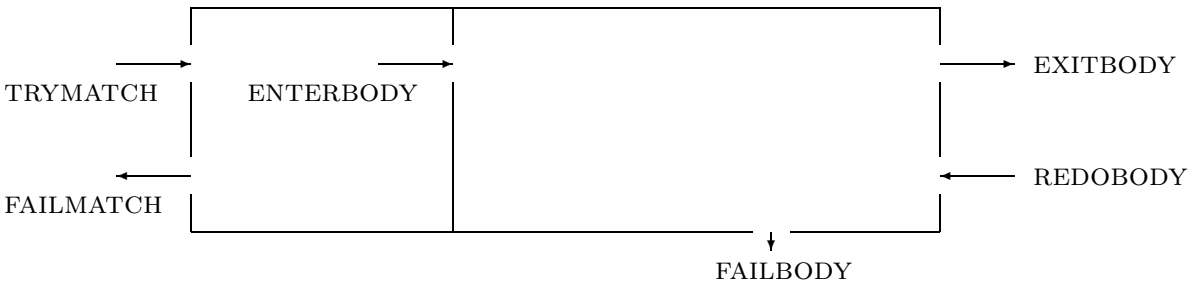
The `error` port has a special status which you can activate or deactivate yourself by means of your own exception handling. You cannot temporarily disable the `error` port by means of the debugger's execution control commands (`skip, next`). You will find debugger commands described in the *IF/Prolog Reference Manual* [1].

The clause boxes inside the goal box correspond to the sequence of ORed (alternative) clauses in the matching predicate.



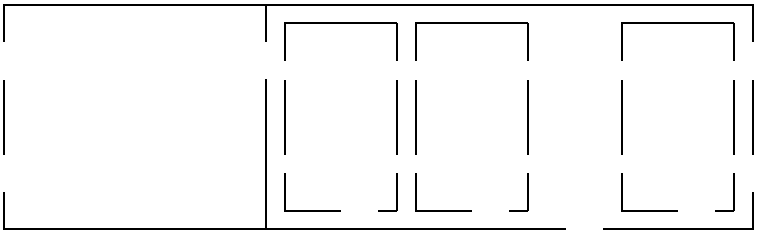
Clause boxes in the goal box

In turn, each clause box contains a left-hand and a right-hand subbox belonging to the clause head and the clause body, respectively. A clause box has the following ports: `trymatch`, `failmatch`, `enterbody`, `exitbody`, `failbody` and `redobody`.



Clause box

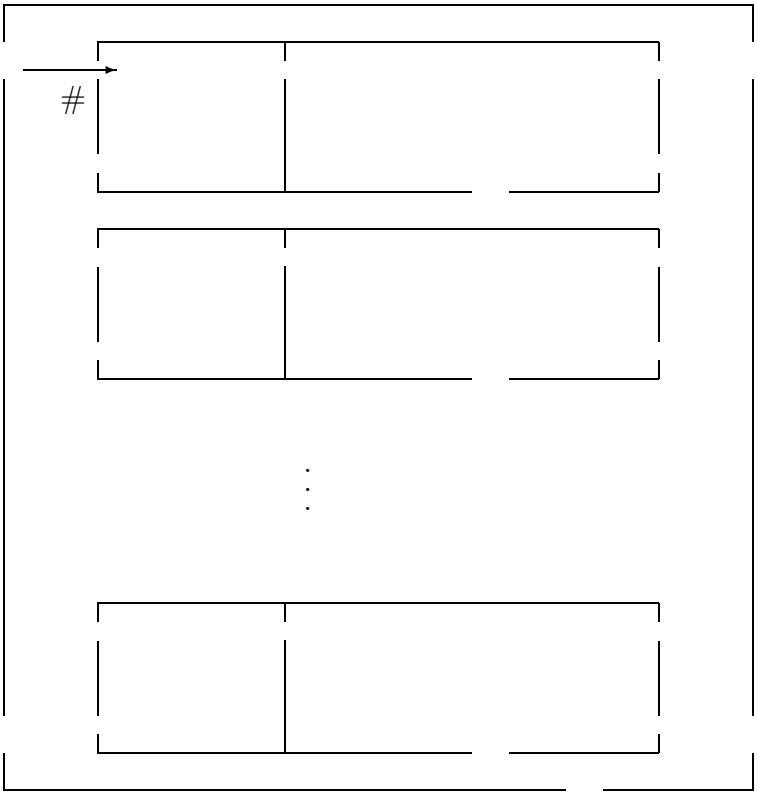
A series of subboxes extending from left to right in the rightmost part of the clause box corresponds to an equal number of ANDed conditions in a clause body. These subboxes are themselves goal boxes with corresponding ports and an internal structure. They are created in the course of goal execution and may be destroyed later.



Subgoal boxes

In the box model, a fact is treated as a special case of a rule. The rightmost part of the clause box contains the box belonging to `true/0` as its only subbox.

The creation of a goal box belonging to the matching predicate corresponds to the first call of a (sub)goal. This goal box contains all clause boxes belonging to the predicate. The path leads from the `call` port of the goal box to the `trymatch` entrance of its first clause box. This clause box is marked with the # character.



Calling a goal

i The Prolog system enters all the available clauses in the goal box at the time of the first call (`call` port). Modifications to the clause set of the predicate after the creation of the goal box have no effect on this goal box. Therefore, if you delete (`retract` predicates) or add (`assert` predicates) clauses after the goal box was created, this has no effect on existing goal boxes.
This is referred to as the **logic update view**of the database.

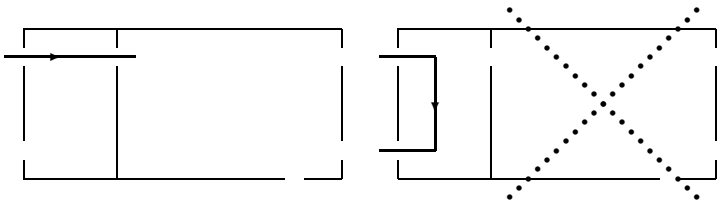
3.5.3 Unification attempt

The Prolog system now checks whether the head of the current clause can be unified with the goal. If it can, the subgoals of the clause may now be processed. If not, Prolog must continue the search for a unifiable clause.

Representation in the box model:

The starting point is the `trymatch` port.

- If unification is successful, Prolog proceeds to the `enterbody` port.
- If not, Prolog leaves the clause box through the `failmatch` exit and destroys the clause box.



Unification attempt

3.5.4 Searching for the next clause

If a goal cannot be successfully proved with the currently selected clause of a predicate, the system attempts to prove it with the next clause of the same predicate. It locates this clause using the previously set choice point. This will take place in two cases:

- when the clause head of the currently selected clause cannot be unified with the goal term, or
- if any of the subgoals of the currently selected clause cannot be proved.

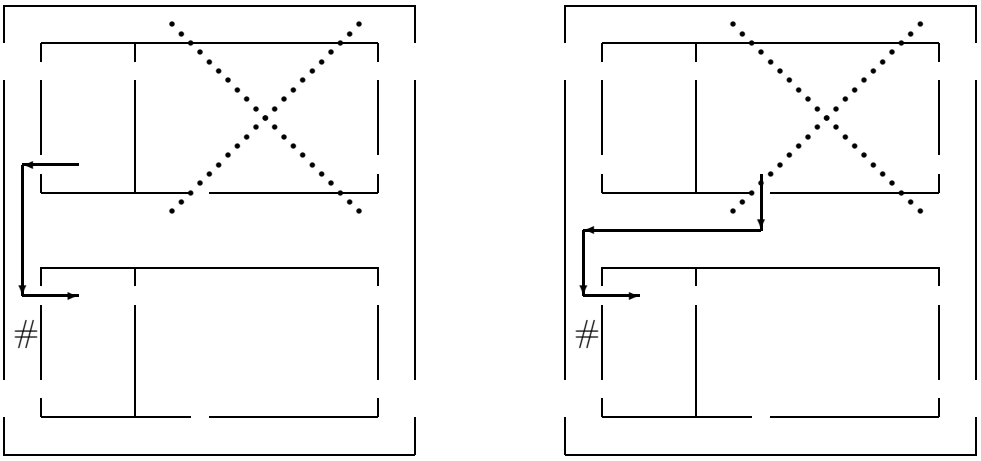
If the selected clause is not the final clause in the predicate, a choice point is set.

Representation in the box model:

Prolog can proceed along two paths from the currently selected clause box (marked with #) to the `trymatch` entrance of the next clause box:

- from the `failmatch` exit of the marked box.
- from the `failbody` exit of the marked box. The `failbody` exit is reached during backtracking.

Once the `trymatch` entrance has been reached, the mark is set to the clause box just entered and the previous box is deleted.



Searching for the next clause

3.5.5 Proving subgoals

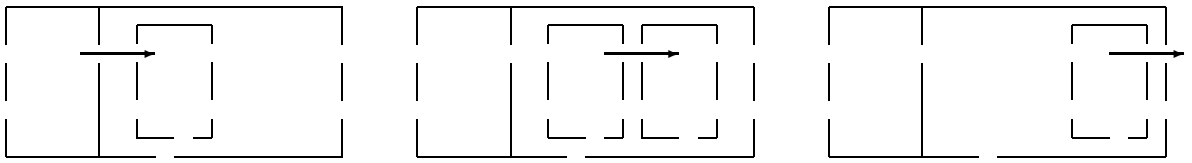
The ANDed conditions of the clause body are understood to be subgoals, which in turn must be proved in the order of their appearance. Any previously performed variable instantiations are taken into account.

Representation in the box model:

During execution, subgoal boxes are entered and left by the following paths:

- From the `enterbody` port, the first subgoal box is created and entered through its `call` entrance.
- At the `exit` port of a subgoal box, the next subgoal box is created, provided another subgoal follows in the clause. Prolog then goes to the `call` port of the new box.

- Prolog goes from the `exit` port of the final subgoal box in a clause box to the `exitbody` exit of the clause box. If the clause box happens to be empty, i.e. all subgoal boxes were destroyed in the course of this path (see section "Proving a goal"), then it too is destroyed.



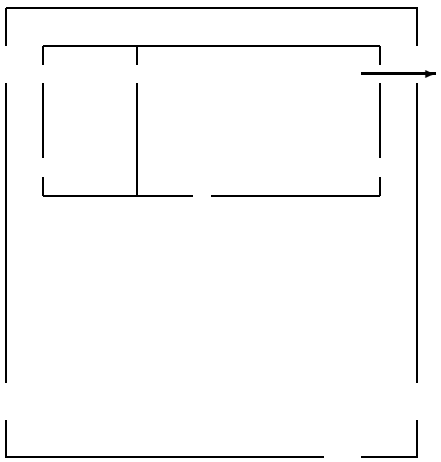
Subgoals

3.5.6 Proving a goal

Once all ANDed subgoals of a goal have been proved, the goal itself is considered to be proved.

Representation in the box model:

Prolog can only reach the `exit` exit of a goal box from the `exitbody` exit of one of its clause boxes. If a goal box is left through the `exit` port and it contains no further clause boxes, it is destroyed.



Success

3.5.7 Failing to prove a goal

A goal fails to be proven when there is only one more clause available for the attempt to prove it, and

- the clause head cannot be unified with the goal term or
- one or more of the corresponding subgoals cannot be proved.

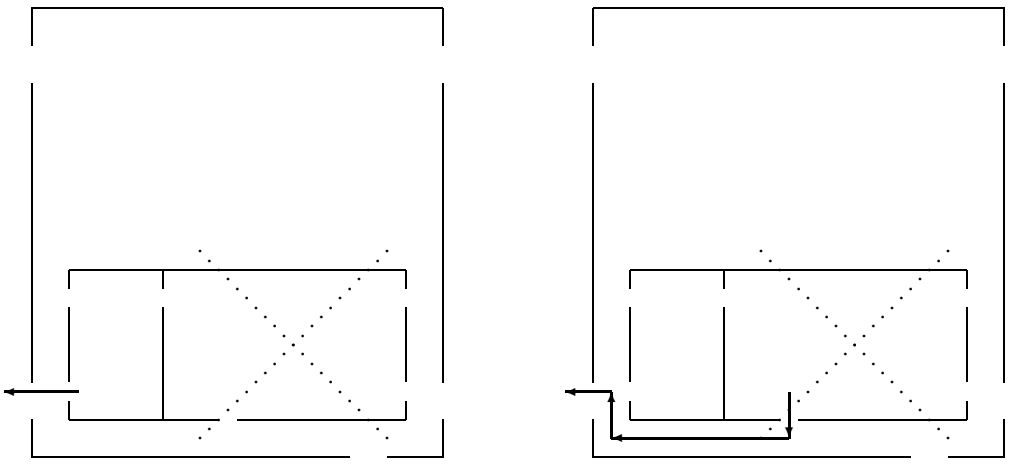
All variable instantiations made while executing this goal are undone.

Representation in the box model:

Each of the two paths to the `fail` exit of a goal box originate from its final clause box:

- the first stems from the `failmatch` exit of the clause box,
- the second stems from the `failbody` exit.

Any goal box left by its `fail` exit is destroyed.



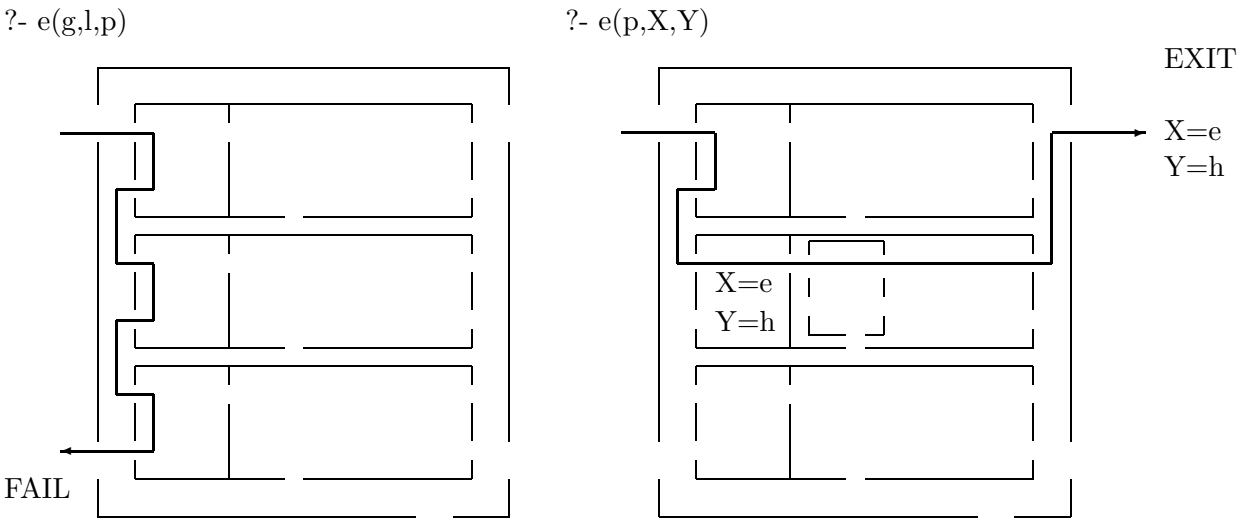
Failure

Example

The goal `e(g,l,p)` cannot be proven in the database containing the predicate `e/3` with the clauses

```
e(a,l,p) :- true.
e(p,e,h) :- true.
e(g,e,h) :- true.
```

because there is no unifiable clause head. The goal `e(p,X,Y)` can be proven with the second clause because the associated clause body is always `true`.



calling e(g,l,p) and e(p,X,Y)

3.5.8 Activating backtracking

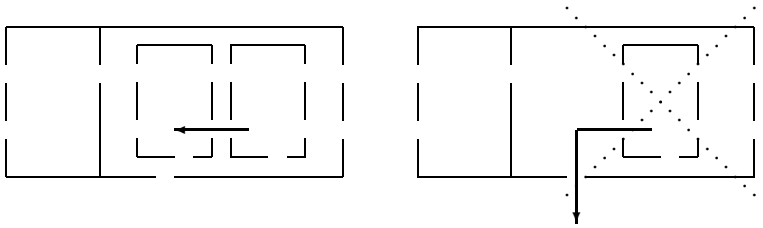
If a subgoal of a clause cannot be proven with the specified variable instantiations, the variable instantiations specified during the execution of this subgoal will be undone and backtracking will be activated. The system returns to the most recent choice point. There are two ways that this can happen:

- The system goes back to a preceding subgoal of the same clause where there are still choice points. The variable instantiations performed since this subgoal was proved are undone.
- Otherwise the clause itself fails.

Representation in the box model:

From the `fail` exit of a subgoal box Prolog proceeds

- to the `redo` entrance of the preceding subgoal box
- or to the `failbody` exit of the surrounding clause box. Once the clause box is left through the `failbody` port it is destroyed.



Activate backtracking

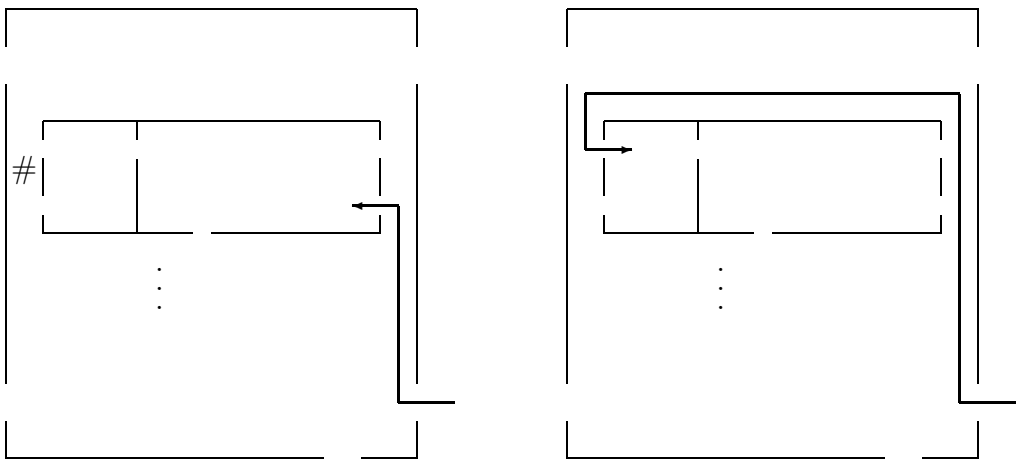
3.5.9 Redoing a goal

When backtracking, the system returns to the last choice point set during execution of the goal. This can also be a choice point in a subgoal. If no such choice point exists, the system returns to the last choice point set when executing the preceding goals.

Representation in the box model:

Three paths begin at the `redo` entrance of a goal box:

- If the goal box still contains a marked clause box, the clause box is entered through the `redobody` entrance.
- If the uppermost clause box is not marked, Prolog enters through its `trymatch` entrance.
- If the goal box is empty, Prolog leaves through the `fail` exit.



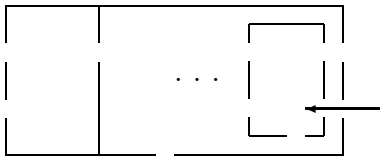
Redoing a goal

3.5.10 Redoing a clause

If, during backtracking, there are still choice points in the subgoals belonging to a clause, Prolog returns to these choice points.

Representation in the box model:

The only path from the `redobody` entrance of a clause box leads to the `redo` entrance of the rightmost goal box contained therein.



Redoing a clause

Example

Extend the above database with the clauses

```
e(a,l,p) :- true.
e(p,e,h) :- true.
e(g,e,h) :- true.
```

by adding the clause

```
g(X,Y) :- e(X,M,V), e(Y,M,V), X \= Y.
```

To prove the goal `g(A,B)`, Prolog now searches in the database for the first clause head with arity 2 and functor `g` which is unifiable with the query. The search sequence thus corresponds to the sequence of the clauses in the database.

The located clause has the head `g(X,Y)`; unification causes the following variable instantiations to take place:

```
X = A
Y = B
```

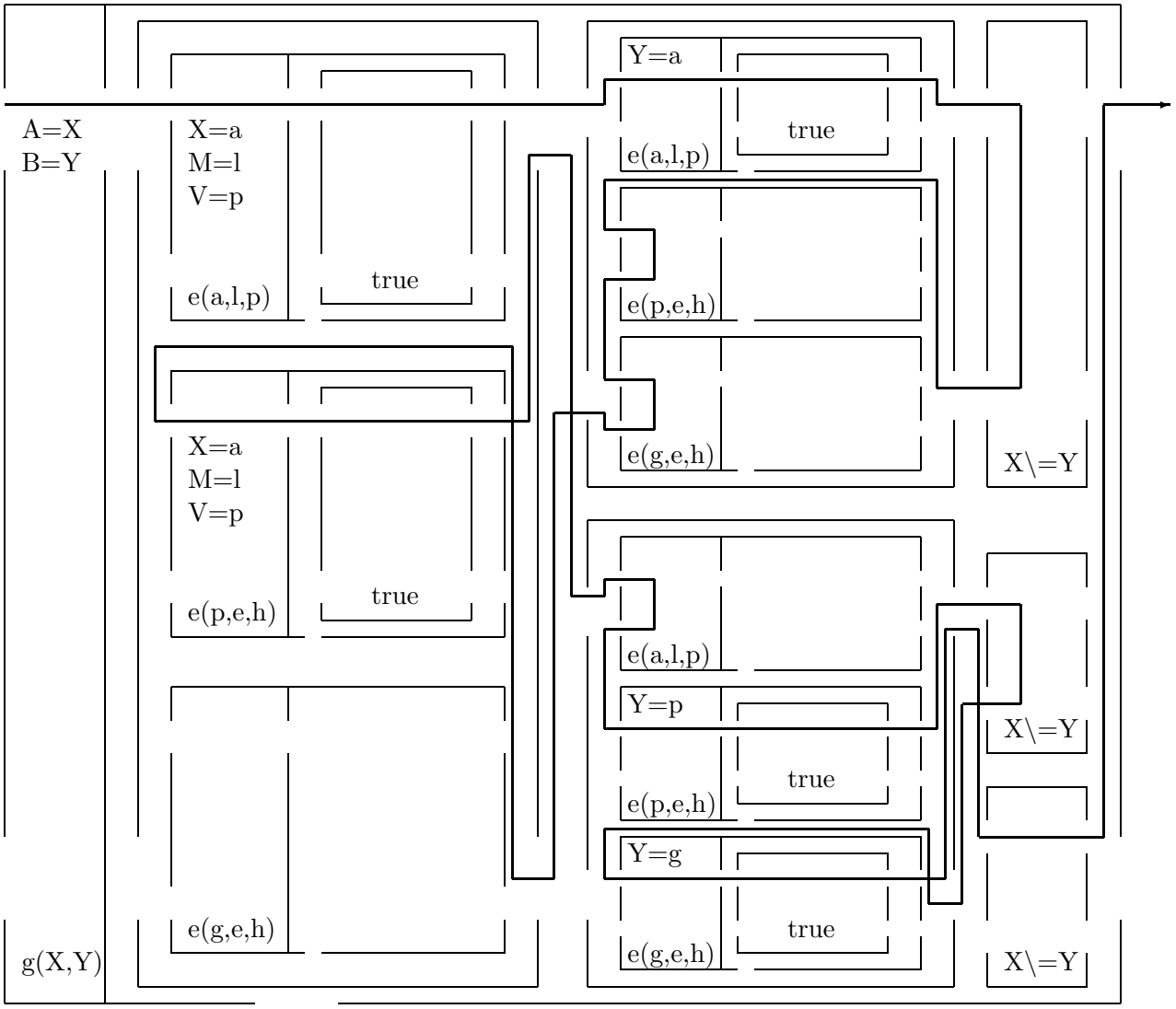
The clause `g(X,Y)` contains three subgoals, which must also be proved. To prove the first subgoal (SG1) the clauses of predicate `e/3` located in the database are searched for the first clause head which is unifiable with
SG1 (= `e(X,M,V)`).
SG1 is unified with the fact `e1`, which is then marked internally as a choice point (CP11) for the first subgoal.

When executing the second subgoal (SG2) the system proceeds in exactly the same way. SG2 is likewise unified with the fact `e1`, and `e1` is marked as a choice point (CP21) for the second subgoal.

The third subgoal cannot be proved with the previously defined variable instantiations, and causes a `fail`. This `fail` triggers the backtracking process internally. The unification performed at the last choice point (CP21) is undone and the system searches for alternatives for executing the second subgoal.

The search for alternatives to SG2 again causes a `fail`, which in turn triggers backtracking. The unification performed at the last choice point (in this case CP11) is undone and the system searches for alternatives for executing the first subgoal, and so on.

When the system finds a solution, it displays the corresponding instantiations in the Prolog input loop. You can interactively activate backtracking (by entering a semicolon '**;**') to search for further solutions.



calling g(A,B)

3.6 Procedural predicates

The concept of a purely logical programming language may look eminently desirable in theory, but in practice it would be too limited. Even the Prolog input/output predicates are difficult to fit into this purely logical scheme because of their side effects.

Moreover, Prolog provides predicates which lead to an irrevocable departure from the purely logical level of programming. The programs can then be understood only procedurally. The most important members of this category are the predicates `!/0` **(cut)**, `fail/0`, `repeat/0` and `for/3`. These are described in greater detail in the following sections.

3.6.1 The cut (!/0)

The predicate `!/0` succeeds only once. If a cut (`!/0`) is called as a subgoal when executing a goal *Goal*, all choice points which were set while executing *Goal* will be destroyed. This has the following effects:

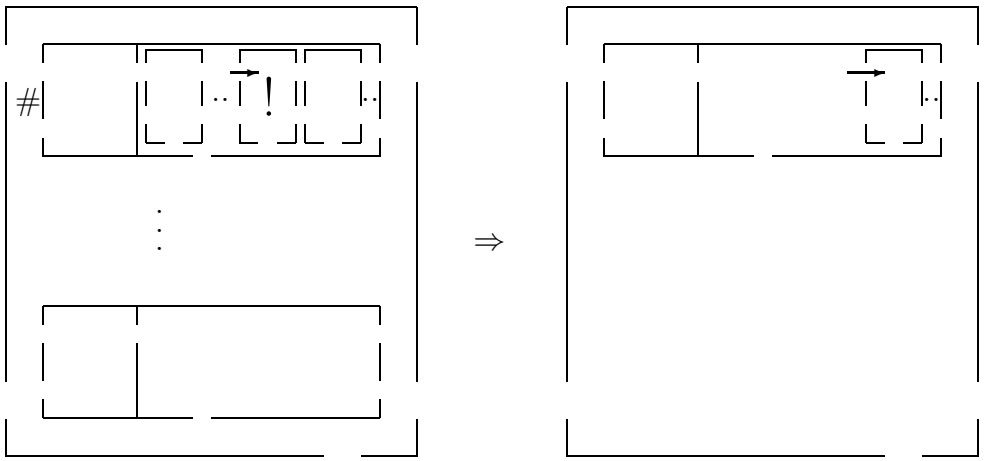
- When executing *Goal*, Prolog will now be committed to the clause in which the cut was called as a subgoal.
- All instantiations made up to this point while executing *Goal* will be made permanent.

If the **cut** is encountered again from the right in the course of backtracking, the proof of the calling *Goal* will fail altogether, since no new variable instantiations or clause selections can be made.

Representation in the box model:

Once a box belonging to the **cut** is entered through the `call` port as a subgoal box within the current clause box, the following happens:

- The **cut** box is left through `exit` and destroyed.
- All subgoal boxes which precede the **cut** box in the current clause box are destroyed.
- All clause boxes which follow the current clause box inside the parent goal box are destroyed.



The cut

The cut

A distinction is made between two types of **cut**:

green cuts: These cuts do not limit the set of alternative proofs of a goal by a program. On the one hand, they enable the programmer to remove from the search tree of a goal those branches which would end with `fail` anyway. On the other hand, if two or more branches in the search tree lead to a proof of a goal with identical variable instantiations, some of these branches can be pruned with green cuts. This reduces the size of the search area to be covered during backtracking, thereby speeding the search up and reducing memory requirements.

red cuts: These cuts limit the set of alternative proofs of a goal by a program. The sequence of clauses in the database is relevant for the set of successful proofs of a goal.

Example

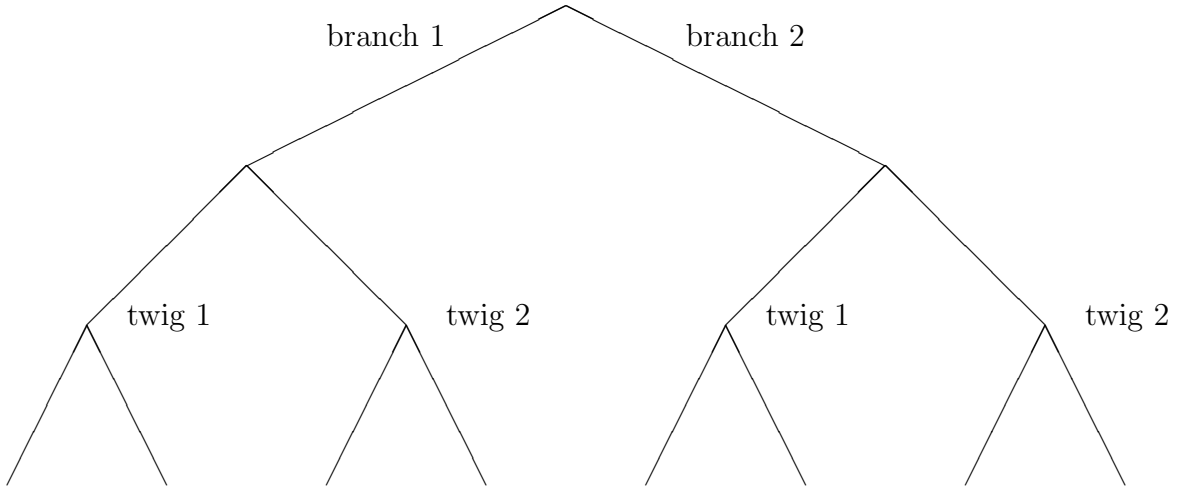
In the first example the following clauses are contained in the database:

```
tree(Branch,Twig,Leaf) :- branch(Branch,Twig),leaf(Leaf).% Clause 1
branch(Branch,Twig) :- branch(Branch),twig(Twig).        % Clause 2
branch(branch1) :- true.                                 % Clause 3
branch(branch2) :- true.                                 % Clause 4
twig(twig1) :- true.                                     % Clause 5
twig(twig2) :- true.                                     % Clause 6
leaf(leaf1) :- true.                                     % Clause 7
leaf(leaf2) :- true.                                     % Clause 8
```

You then enter the following query:

```
[user] ?- tree(Branch,Twig,Leaf).↩


Branch  = branch1
Twig    = twig1
Leaf    = leaf1 ;

Branch  = branch1
Twig    = twig1
Leaf    = leaf2 ;

......

Branch  = branch2
Twig    = twig2
Leaf    = leaf2 ;

no
```

In response, you are given all paths in the tree from the trunk to any leaf. Originally, then, the tree has the form:

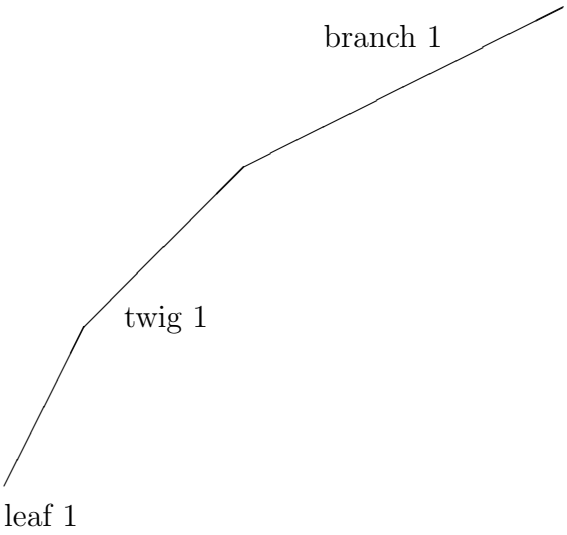


Original form of the tree

Using a **cut**, the tree should now be pruned. First add it to the end of clause 1, i.e.:

```
tree(Branch,Twig,Leaf) :- branch(Branch,Twig),leaf(Leaf),!.
                                              % Clause 1-new
```

The **cut** removes all alternatives for the subgoals located to its left. Now you are only given the first of the solutions created above; the tree therefore has the structure.

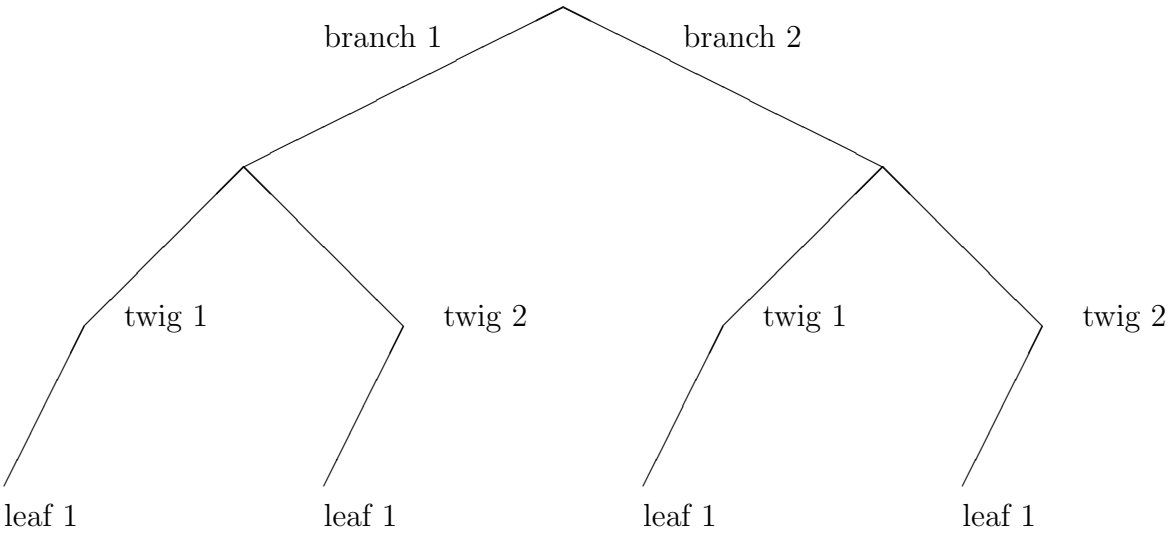


Tree after rigorous pruning

Prune the tree very carefully by applying the **cut** only to the first clause of the predicate `leaf/1`, i.e. to clause 7.

```
leaf(leaf1) :- true,!.  % Clause 7-new
```

This removes the solutions for `leaf/1` which result from clause 8, thus yielding the following form:



Tree after careful pruning

All branches and twigs are still there.

Example

Red cuts can be used to replace subgoals which may under certain circumstances be difficult to formulate. This is illustrated below by a program for the predicate `remove(`*Xs*,*X*,*Ys*`)` which removes all list elements *X* from an input list *Xs* and unifies the remainder with *Ys*.

Without using a **cut**, you can create this predicate as follows:

```
/* Program 1 */

remove([X|Xs1], X, Ys) :-                       % Clause 1
        remove( Xs1, X, Ys).

remove([Y|Xs1], X, [Y|Ys1]) :-                  % Clause 1
        Y \== X,
        remove(Xs1, X, Ys1).

remove([],X,[]).                                % Clause 1.3
```

The condition `Y \== X` is essential as a subgoal in clause 1 because otherwise, during backtracking or when interchanging the clauses, result lists will be created which still contain the list element `X`. The fact that the only promising clause in the form *[X | Xs1]* of the input list is clause 1 of program 1 can be highlighted by inserting a **cut** at the start of its body.

```
remove([X|Xs1], X, Ys) :-                       % Clause 1
        !,
        remove(Xs1, X, Ys).
```

This prevents an attempted proof with the final two clauses from taking place during backtracking, since it is bound to fail anyway. The logical behavior of the predicate is not changed externally, but in certain cases processing will be expedited. In other words, we are dealing with a **green cut**.

Admittedly, following this modification you can delete the condition `Y \== X` in the second clause. This then gives you

```
/* Program 2 */

remove([X|Xs1], X, Ys) :-                       % Clause 2
        !,
        remove(Xs1, X, Ys).
```

```
remove([Y|Xs1], X, [Y|Ys1]) :-                % Clause 2
        remove( Xs1, X, Ys1).

remove([],X,[]).                              % Clause 2.3
```

In Program 2 you can no longer remove the **cut** without changing the meaning. Likewise, exchanging the first two clauses will cause a change of meaning. In other words, we are now dealing with a **red cut**. A further example of this can be found on page 93.

3.6.2 Loops in Prolog

repeat/0

The predicate `repeat/0` can be defined as follows:

```
repeat :- true.
repeat :- repeat.
```

`repeat/0` is always successful, and thus creates an unlimited number of choice points for backtracking. Subgoals of a clause called prior to `repeat/0` will therefore never be reached when backtracking within this clause.

This predicate is often used in conjunction with the following `fail/0` to create loops.

fail/0

Proof of `fail/0` will invariably fail.

Example

The clause

```
endless :- repeat, fail.
```

creates an endless loop.

You can implement conversion to a repeat-until loop with a defined termination by means of non-logical predicates. Possible ways are to

- use input/output predicates
- manipulate the database
- use a global variable.

Example

One example of input/output predicates is the (incomplete) simulation of a Prolog input loop by means of the following program:

```
dialog :-
        repeat,
        prompt,
        read( Goal),
        work( Goal),
        Goal == end,
        !.

prompt :-
        write('?-').

work(end) :-
        true.

work(Goal) :-
        Goal,
        !,
        write('yes'),
        nl.

work(Goal) :-
        write('no'),
        nl.
```

Calling `dialog` starts the loop and entering `end.` terminates it. The **cut** in the dialog clause prevents the repeat loop from being run through again in the course of backtracking after the end criterion *Goal*`==` `end` has been reached. The **cut** in the second `work` clause replaces the condition `not` *Goal* in the final clause and thus prevents double execution of *Goal*. Both `cuts` are **red cuts**.

Example

To illustrate manipulation of the database we will take a count predicate which, when called, displays the integers between *Min* and *Max* on the current output. When global variables are used, the predicates are specified as alternatives following the comment character.

```
count( Min, Max) :-
        Min > Max,
        ! .
```

```
count( Min, Max) :-
        asserta( number( Min)),     % set_global(number,Min)
        repeat,
        retract( number( Number)),  % get_global(number,Number)
        write( Number), nl,
        Y is Number + 1,
        asserta( number( Y)),       % set_global(number,Y)
        Number == Max,
        !,
        retract( number( Y)).       % unset_global(number)
```

The first clause prevents the program from going into an endless loop.

You can also create loop-like execution structures using recursive predicates. Repeat-fail loops require less memory than recursions, unless you are able to put the recursive predicate into tail-recursive form. Tail-recursion (i.e. the recursive call is located at the end of the clause) is helpful because the Prolog system generates particularly efficient code for these clauses.

With recursive constructions, too, you must pay attention to the way you formulate the tail condition so that you do not produce an endless recursion.

Example

The function of the predicate `count/2` given above can also be performed by an tail-recursive predicate `count_rec/2` with the following clauses:

```
count_rec( Min, Max) :-
        Min > Max,
        ! .

count_rec( Min, Max) :-
        write( Min), nl,
        Min_New is Min + 1,
        count_rec( Min_New, Max).
```

for/3

Loops in which, basically, a series of integers is created and utilized can be conveniently created using the built-in predicate `for/3` (see description in the Reference section on built-in predicates in the *IF/Prolog Reference Manual* [1]).

Example

`for/3` provides the easiest way to create the count predicate:

```
count_for( Min, Max) :-
        Min > Max,
        ! .

count_for( Min, Max) :-
        for( Min, Number, Max),
        write( Number), nl,
        Number == Max.
```

Chapter 4

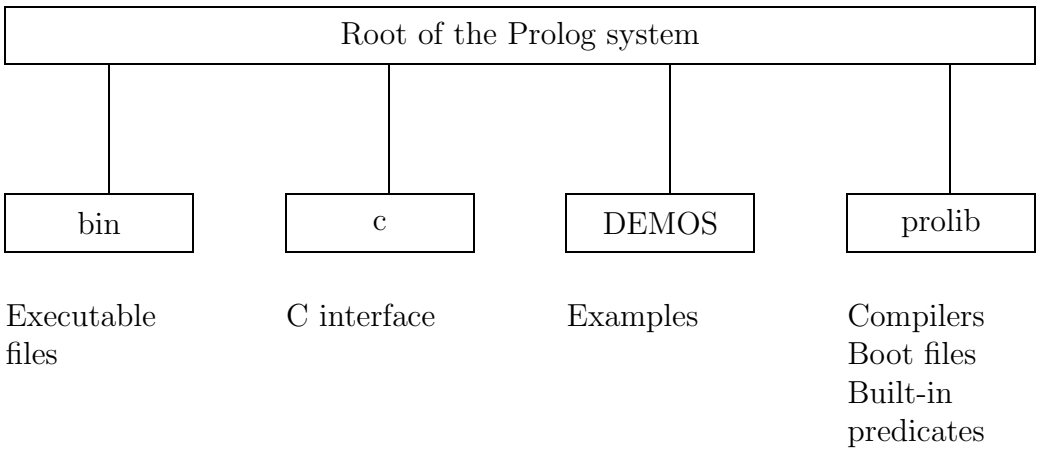
The Prolog system

This chapter describes the Prolog system and its environment. Topics dealt with include the organization of the Prolog system, calling and exiting Prolog, and the different system modes. It describes how you can query diverse information about the Prolog system and provides an overview of the interfaces of the Prolog system.

As IF/Prolog is available on different platforms with different operating systems, input key or path specifications may be different for your system. System-specific settings and special considerations are described in the release notice for your Prolog system.

4.1 Organization of the Prolog system

The root directory of the Prolog system is defined at installation time. The default setting for the root directory is operating system dependent. If you copy an existing Prolog system to a different position in the file system, you must assign the pathname of the new root directory to the environment variable `PROROOT`. The various components of the Prolog system are stored under the root as follows:



4.1.1 Calling Prolog

When calling Prolog you can specify two different types of options: **system arguments** which have a Prolog-specific meaning, and **user arguments** which you have defined yourself. You must enter system arguments first; they can be given in any order. You can specify user arguments after the system arguments. If system arguments are included among the user arguments, they will no longer be recognized as such and will be treated as user arguments.

You call the Prolog system with the command `prolog`.

```
prolog [-option [systemargument ]] ... [userargument] ...
```

`-option` [*systemargument*] Prolog offers a range of built-in options which allow you to define set values for system arguments. With these system arguments you can e.g. specify the files which are to be loaded into the database when the system is started, or specify values for initialization of the Prolog system. In the latter case the Prolog system may round the specified values to useful values. The maximum values depend on the size of the available main memory. Initialization is not normally required, however, because the Prolog system automatically matches its memory usage to the system requirements. The built-in predicate `system_parameters/1` allows you to query the system options and their associated arguments which you have specified in the call, including the system name `prolog`. The meanings of the various system arguments are explained in the following.

`-debug` Sets the Prolog flag `debug` to `on`. This means that all the predicates which you read into the database with `consult/1, reconsult/1` or `assert/1/2` are implicitly declared as dynamic, thereby supporting the debugging of a program.

`-iso` Only the facilities required by the ISO standard for Prolog are available to you. You cannot use the extensions specific to IF/Prolog.

`-language` *charset* Sets the character set, which will be supported by the input/output built-in predicates of IF/Prolog. See also the *IF/Prolog Reference Manual* [1], predicate `current_language/2`. The parameter *charset* may have one of the following values:

- `ascii` This is the default value. It specifies that I/O predicates accept and deliver ASCII code text.
- `euc` The **EUC** character set is supported.
- `sjis` The **SJIS** character set is supported.
- `chinese` The multi-octet set of Chinese characters is supported.
- `korean` The multi-octet set of Korean characters is supported.
- `taiwanese` The multi-octet set of Taiwanese characters is supported.

`-mi` *number* Sets the minimum size of the dynamic memory blocks requested by the Prolog system. *number* is an integer specifying the size in bytes of the memory made available. For each 1024-byte unit you may also use the abbreviation 1k or 1K, e.g. 10k instead of the value 10240.
Default: 8K

`-ms` *number* *number* bytes are made available as the initial size of the total memory for the Prolog system.
Prolog then partitions the individual memory areas itself. *number* is an integer specifying the size in bytes of the memory made available. For each 1024-byte unit you may also use the abbreviation 1k or 1K, e.g. 10k instead of the value 10240.
Default: 8K

`-msc` *number* *number* bytes are made available as the initial value of the constraint data stack.
number is an integer specifying the size in bytes of the stack made available. For each 1024-byte unit you may also use the abbreviation 1k or 1K, e.g. 10k instead of the value 10240.
If required for program execution, the memory area is automatically increased.
Default: 8K

`-msg` *number* *number* bytes are made available as the initial value of the global data stack.
number is an integer specifying the size in bytes of the stack made available. For each 1024-byte unit you may also use the abbreviation 1k or 1K, e.g. 10k instead of the value 10240.
If required for program execution, the memory area is automatically increased.
Default: 30K

`-msl` *number* *number* bytes are made available as the initial value of the local data stack.
number is an integer specifying the size in bytes of the stack made available. For each 1024-byte unit you may also use the abbreviation 1k or 1K, e.g. 10k instead of the value 10240.

If required for program execution, the memory area is automatically increased.
Default: 30K

`-mst` *number* *number* bytes are made available as the initial value of the variable instantiations stack (trail).
While the system is attempting to prove a goal, the stack is used to store information about current variable instantiations which will have to be cleared again ("undone") during backtracking.
number is an integer specifying the size in bytes of the stack made available. For each 1024-byte unit you may also use the abbreviation 1k or 1K, e.g. 10k instead of the value 10240.
If required for program execution, the memory area is automatically increased.
Default: 8K

`-msx` *number* *number* bytes are made available as the initial value of the extended terms stack.
number is an integer specifying the size in bytes of the stack made available. For each 1024-byte unit you may also use the abbreviation 1k or 1K, e.g. 10k instead of the value 10240.
If required for program execution, the memory area is automatically increased.
Default: 8K

`-nobuf` The standard input streams are not buffered by Prolog.

`-nonotify` Sets the Prolog `consult` flag to `nonotify`. This means that no messages are displayed on the screen when a file is loaded.

`-nosignal` Prevents the Prolog system from managing signals itself, i.e. it defines no user-defined signal handling routines. The predicate `signal_control/2` has no effect.

`-notty` The standard streams `screen` and `keyboard` are not opened, but are mapped to the standard input and standard output. In this case it is not possible to use the keyboard and the screen as the input/output devices when the standard input/output has been redirected.

`-nowarnings` Sets the Prolog `warnings` flag to `off`. This means that when `consult/1` is called no warnings will be displayed on the screen.

`-program` *Goal* If *Goal* is a syntactically correct goal, it will be invoked automatically inside the module `user` after the system start. After successful execution of that goal Prolog is terminated. If *Goal* is incorrect, Prolog is terminated immediately. It is possible to qualify *Goal* with the name of a loaded module, where it is to run.

`-prompt` Sets the Prolog `prompt` flag to `on`. This means that Prolog will output a prompt, or that backtracking can be initiated interactively with a semicolon, even if terminals are not used for standard input/output.

`-root` *pathname* Sets the pathname under which the files of the Prolog runtime environment can be accessed. This option takes precedence over the environment variable `PROROOT`.

`-sp` *path* Sets the initial value of the Prolog `search_path` flag to the list specified in *path*. Pathnames are separated by means of a colon (:) (UNIX) or semicolon (;) (DOS).

`-stream` Prolog is started on an alpha screen (no link to OSF/Motif). The debugger is started without a graphical user interface (see also `debug_config/3`).

`-xenv` Starts the graphical development environment under OSF/Motif.

`--` Identifies the end of the system arguments. All following arguments are considered to be user arguments, even if the name is the same as that of a system argument.
The end of the system arguments is also considered to have been reached if the command line contains an argument which is not a system argument.

The following options are always evaluated last, regardless of their position in the call line:

`-c` *filename* The Prolog source file *filename* is read into the database with `consult/1` (after the start file, if there is one; see `-st` option).
The system searches first the current directory and then the directories specified under `search_path`, and proceeds in the following order:

- for a file *filename*
- then, if none is found, for a file *filename.pro*

The contents of *filename* must be a syntactically correct Prolog source text.

`-l` *filename* The compiled Prolog file *filename* is loaded into the database (after the start file, if there is one; see `-st` option).
The system searches first the current directory and then the directories specified under `search_path`, and proceeds in the following order:

- If the file *filename* contains the suffix *.cmp*, it is loaded into the database.
- Otherwise the system searches for the file *filename.cmp* and loads it.

i You may also specify the options `-c` *filename* and `-l` *filename* more than once with different files. If you do, they are evaluated in the specified order.

`-r` *file* The system status is loaded from *file* which was saved with `save_system/1`. If Prolog cannot open *file* or if *file* does not have the suffix *.pst*, the suffix is appended and the name thus produced is used.

`-st` *startfile* If a hyphen (-) is specified for *startfile*, then no start file is loaded or consulted. Otherwise the specified start file will be the first file to be loaded or consulted when the Prolog system is called. The system searches the current directory for the file in the following order:

- If *startfile* has the suffix *.cmp*, the system loads this file.
- If *startfile* has the suffix *.pro*, the system consults this file.
- If the name has no suffix, the system
  - first searches for and loads a file called *startfile.cmp*;
  - if no such file is found, it attempts to consult a file called *startfile*;
  - then, if this file also does not exist, it consults the file *startfile.pro*.
  - Default (option `-st` not specified):
    Prolog checks
    * whether the environment variable `PRORC` is set and it it is, tries to load the file with this name.

    Otherwise, the system searches the `HOME` directory
    * for a file called *.prorc.cmp*; if the file exists, it is loaded.
    * for a file called *.prorc*; if the file exists, it is consulted.
    * for a file called *.prorc.pro*; if the file exists, it is consulted.

i If one of the files which you specified in the options `-st`, `-c` or `-l` could not be found, an appropriate exception is generated and the Prolog process is terminated.

userargument You can define your own arguments and specify them in accordance with your own syntax rules when calling an extended Prolog system. These arguments are called user arguments, i.e. you determine the form in which the arguments are specified (e.g. with or without options) and what effect the arguments have in your Prolog application. In Prolog you can query the user arguments specified in the call by means of `user_parameters/1`.

Example

1. Calling without user arguments

```
$ prolog -c test
```

Prolog is called with the default values and the start file, if one is available is read in. The Prolog source file *test*, if it exists, is then read into the database. If it is not found, the file *test.pro* is read in instead.

```
$ prolog -ms 1000k -mst 200k -l userpred
```

Prolog is called with new values for database size and trail size. If available, the start file is read in. Next the file *userpred.cmp*, if found, is loaded into the database.

2. Calling with user arguments

Suppose that you want to read test data from a file in a test run. For this purpose, you can define your own option to enable you to specify the name of the file containing the test data. In the example, we shall call the option *-data*.

```
$ prolog -ms 10000k -mst 200k -c students -data test

IF/Prolog ...
Copyright (C) Siemens Nixdorf Informationssysteme AG, 1996

*** consult 'students.pro': loaded in 0.03 sec.
```

In a Prolog session, you can query the arguments in several different ways.

```
[user] ?- system_parameters(L).↩


L       = ['/opt/lib/prolog/bin/prostd','-root','/opt/lib/prolog','-ms',
'10000k','-mst','200k',' -c',students]

yes

[user] ?- program_parameters(L).↩


L       = ['/opt/lib/prolog/bin/prostd','-root','/opt/lib/prolog','-ms',
'10000k','-mst','200k',' -c',students,'-data',test]

yes
```

```
[user] ?- user_parameters(L).↩

L       = ['-data',test]

yes
```

You can read the file containing the test data by entering, for example, the following predicate:

```
read_data:-
        user_parameters(L),
        append(_,['-data',File|_],L), !,
        see(File),
        read_data,
        seen.
```

3. Calling with a start file

The start file offers a convenient means of changing default values or defining useful, frequently used predicates. For example, if the file *.prorc* has the following contents:

```
:- init(default_editor(_,emacs)).
writeln(Term) :- write(Term), nl.
```

then `emacs` is immediately selected as the file editor for the predicates `edit/0` and `edit/1` when the Prolog system is started. Also, the predicate `writeln/1` is automatically available; it outputs a term and a line feed each time a call is made.

4.1.2 Exiting Prolog

You can either exit Prolog temporarily or quit Prolog by ending the session.

Predicates for making a temporary exit

Predicate	Purpose
system	Switch to operating system level
system(+*Command*)	
system(+*Command*, ?*ExitStatus*)	Execute operating system command
system(+*Command*, ?*Input*, ?*Output*)	
edit *[@ Module]*	Edit a file
edit(+*Filename*) *[@ Module]*	Edit a file

With the predicate `system/0` you remain at the operating system level until you press the key combination CTRL D or enter the `exit` command to bring up the Prolog system prompt again. With all other predicates, control returns automatically to Prolog when execution of the command is completed.

Predicates for terminating a Prolog session

Predicate	Purpose
halt	Terminate Prolog
halt(+*ExitStatus*)	Set exit status and terminate Prolog

`halt/0` and `halt/1` always return you to the operating system level; `end_of_file/0` or the key combination CTRL D return you to operating system level only if you are on the highest Prolog top level (see predicate `break/0`).

4.2 Querying information about the system

Information about the current system configuration can be queried while a Prolog session is in progress.

4.2.1 Querying system size parameters

The predicate `is/2` allows you to query the system size parameters:

```
[user] ?- X is ssize.↩
```

System time	Meaning
cputime	Seconds of CPU time used
time	System time in seconds

System size	Meaning
csize	Total size of local stack (bytes)
cused	Local stack in use (%)
dsize	Total size of database (bytes)
dused	Database in use (%)
esize	Total size of extended term stack (bytes)
eused	Extended term stack in use (%)
gsize	Total size of global stack (bytes)
gused	Global stack in use (%)
tsize	Total trail stack size (bytes)
tused	Trail stack in use (%)
ssize	Total stack size (bytes)
sused	Stack in use (%)

The following zero arity functions show the value range available for integers:

System size	Meaning
minint	Smallest representable integer
maxint	Largest representable integer

If the IF/Prolog system has been configured with multiple precision integer support, a much wider range of integers is available. However, the functions above return always the smallest or largest integer that can be represented as one word on the underlying hardware.

4.2.2 Querying system properties

The predicate `current_prolog_flag/2` allows you to query the flags defined in the Prolog system.

```
[user] ?- current_prolog_flag(Flag, Value).↩
```

The following Prolog flags may be queried and modified:

Flag	Meaning	Default value
backtrace	Maximal length of backtrace list on exception	10
char_conversion	Table-controlled character conversion	on
consult	Output message on loading Prolog text	notify
debug	Implicitly declare predicate as dynamic	off
double_quotes	Handling of double quotes	codes
extended_syntax	Input of cyclic terms	off
nested_comments	Allow nested comments in Prolog text	off
prompt	Input prompt and interactive backtracking even if standard input/output is not terminal	off
read_error	Handling of syntax error during analysis with `read_term/2/3` etc.	error
report_read_error	Handling of syntax error during analysis with `parse_atom/6`	on
search_path	Input files are sought in the list of specified directories	`[]`
unknown	Handling of undefined predicates	error
warnings	Output warnings on possible errors with `consult/1` etc.	on
write_depth	Maximal depth of terms output in Prolog toplevel	10

The predicate `set_prolog_flag/2` allows you to change these default values.

The following table gives an overview of the flags which you are not able to modify. The flags themselves provide information about how the Prolog system is implemented or configured. Since the default values differ for different system variants, they are not listed below.

Flag	**Meaning**
bounded	Correct results with integer arithmetic
integer_rounding_function	Rounding up and down for functions and the division remainder `rem`
iso	Restriction to predicates which conform to the ISO draft standard; you can set this flag only when calling the Prolog system
float_max	Maximum representable floating point number
float_min	Minimum representable floating point number
max_arity	Maximum arity of predicates and structures
max_integer	Maximum representable integer
min_integer	Minimum representable integer
notty	Input/output devices are not terminals
signal	Enable/disable signal handling; You can set this flag only when calling the Prolog system

Example

The following example shows the default values of all flags on a MIPS-based system with IRIX 5.3:

```
[user] ?- current_prolog_flag(X,Y), write((X,Y)), nl, fail.↩


backtrace , 10
bounded , true
char_conversion , on
consult , notify
extended_syntax , off
debug , off
double_quotes , codes
float_max , 1.79769e+308
float_min , 4.94066e-324
integer_rounding_function , toward_zero
iso , off
max_arity , 127
max_integer , 2147483647
min_integer , -2147483648
nested_comments , off
notty , off
```

```
prompt , off
read_error , error
report_read_error , on
signal , on
search_path , []
unknown , error
warnings , on
write_depth , 10

no
```

4.2.3 Querying other information

The following predicates also provide information about the system:

Predicate	Purpose
system_name(?*SystemName*)	Query operating system
system_hostid(?*Hostid*)	Query machine identification
prolog_version(?*Version*)	Query Prolog version
prolog_serial(?*Serial*)	Query Prolog serial number
proroot(?*Path*)	Query path of Prolog system

4.2.4 Querying memory management

You can control or check memory management of the Prolog system by using the following predicates:

Predicate	Purpose
current_memory_management(?*Area*, ?*Parameters*, ?*Value*)	Query memory management parameters
garbage_collection(+*Area*)	Activate garbage collection
set_memory_management(+*Area*, +*Parameters*, +*Value*)	Set memory management parameters
statistics	Output system statistics
statistics(*?Type, ?ResultList*)	Query system statistics

For a detailed description of the predicates, refer to the *IF/Prolog Reference Manual* [1].

IF/Prolog manages six memory areas, and also performs garbage collection in those areas as required. You may query the parameters for the memory areas with `current_memory_management/3` and set new parameters with `set_memory_management/3`. You can use `garbage_collection/1` to initiate garbage collection for a specific memory area.

IF/Prolog manages the following memory areas:

Area	Meaning
database	Entire database
constraints	Stack for constraint processing
extension	Stack for term extensions
local	Stack for local variables and choice points
global	Stack for structures
trail	Stack for term instantiations which are to be undone during backtracking

Example

```
[user] ?- statistics.↩


memory statistic :    size      used    #gc collected   gc-time

        database :  962544    490560      0         0      0.00
          global :   30720       496      0         0      0.00
     constraints :    8192         0
           local :   30720       700      0         0      0.00
       extension :    8192        32      0         0      0.00
           trail :    8192         0      0         0      0.00

           total : 1048560    576576      0         0      0.00

            date : Fri Sep  9 14:32:43 1996
         cputime : 1.40

yes
```

You can query or set the following parameters for any of these memory areas:

Parameter	Meaning
minimal_size	Minimum size of memory area in bytes
maximal_size	Maximum size of memory area in bytes
default_gap	Percentage in relation to memory used; indicates the minimum percentage of memory used which must remain free before further memory is allocated.
minimal_gap	Minimum number of bytes which must be free in the memory used after garbage collection before further memory is allocated.
maximal_gap	Maximum number of bytes which must be free in the memory used after garbage collection before further memory is allocated.
trace	Enable/disable output of a message to standard error output in the case of garbage collection.

By using the predicate `statistics/2`, you can query the memory size and the data for garbage collection for all areas by means of backtracking. In the following example the Prolog system has just been called and consequently the list which holds the data for garbage collection does not yet contain any significant values for `garbage_collection`.

Example

```
[user] ?- statistics(X,Y).↩
```

```
X       = time
Y       = [cputime = 1.41] ;

X       = memory
Y       = [system = [1048560,575824],database = [962544,489808],
global = [30720,716],constraints = [8192,0],local = [30720,760],
extension = [8192,32],trail = [8192,0]] ;

X       = garbage_collection
Y       = [database = [0,0,0],global = [0,0,0],constraints = [0,0,0],
local = [0,0,0],extension = [0,0,0],trail = [0,0,0]] ;

no
```

4.3 Environment variables

You can modify the environment with which the Prolog system works. This is done using variables whose contents you may change individually and which are then evaluated accordingly by the Prolog system. The following table summarizes all environment variables used by the Prolog system:

Environment variable	Meaning
DISPLAY	OSF/Motif server name
EDITOR	Editor started with `edit/0/1`
HOME	Directory of user login
LANG	Set language environment
PROHELP	Path for help files
PROPATH	Path for directories in which files to be opened should be looked for
PROROOT	Path for Prolog system files
PRORC	Start file for the Prolog call
SHELL	Command interpreter started with `system/0`
TERM	Name of terminal
TMPDIR	Directory for temporary files

Set environment variables

You set environment variables in the shell. Note that you must specify environment variables in uppercase letters.

Example

Bourne shell:

```
$ EDITOR=emacs; export EDITOR↩
```

C shell:

```
% setenv EDITOR emacs↩
```

If you do not wish to make these inputs again after every login, then you should store them in the start file *.profile* (Bourne shell) or *.login* (C shell). You use `unset` or `unsetenv` to reset the variables.

Environment variable DISPLAY

determines the OSF/Motif server. A display is defined using the name of the host to which the display is connected, the display number and the screen number in the following format:

```
DISPLAY=[hostname]:displaynumber[.screennumber]
```

If the server is located on a different host than the client, this should be specified. Since several servers for several displays can be started on one host, the display number should be specified. Numbering starts with 0. If you are unsure, check with your system administrator for *displaynumber* or *screennumber*. For further information, refer to the *IF/Prolog OSF/Motif Interface* manual [3].

Environment variable EDITOR

defines which editor is used for editing files.

Default: operating system dependent

Environment variable PROHELP

defines the path under which the help files for Prolog are stored.

Default: *$PROROOT/prolib*

Environment variable PRORC

defines the path of the file which is opened when the Prolog system is called and the `-st` option is not set. If you have not set `PRORC` and have not specified the `-st` option when calling Prolog, the Prolog system attempts to open the file *.prorc* in your `HOME` directory.

Environment variable PROROOT

defines the path under which the system files are stored by Prolog. The pathnames may differ on different operating systems. Should the path specified here not be applicable to your system, check in the release notice.

Default: operating system dependent (e.g. */usr/lib/prolog* or */opt/lib/prolog*)

Environment variable SHELL

defines the command interpreter which is called if you temporarily exit the Prolog system. In UNIX, this environment variable is normally already preset to the value contained in the file */etc/passwd*.

4.4 Interfaces of the Prolog system

The Prolog system offers the following interfaces:

Input/output interface

The input/output interface of the Prolog system is structured on the basis of a driver model, enabling you to manage different input/output streams with a specific set of predicates. You can program your own device drivers using the C language and link them into the Prolog system. For a detailed description of the input/output interface, refer to chapter 5.

Active interface to the C language

This interface enables Prolog predicates to be programmed as C functions and then to be used like normal predicates. The active interface to the C language is described in detail in chapter 10.1.

Passive interface to the C language

The passive interface to the C language enables Prolog predicates to be called in a C program in a similar way to C functions and also to initiate backtracking, where appropriate. For further information on this interface, refer to chapter 10.2.

C device drivers

You can use the C device driver interface to implement your own device drivers in C and link them into the Prolog system. For a description of this interface, refer to chapter 10.3.

Interface to OSF/Motif

If the Prolog system has been configured with OSF/Motif and an X server is available, the interface to OSF/Motif is automatically available. OSF/Motif objects such as menus and windows can be created and manipulated with built-in predicates. The built-in predicates for event handling allow flexible configuration of interactive user interfaces. For a description of this interface, refer to [3].

If you wish to use the predicates of the OSF/Motif interface in modules, you must first import the module `motif`.

```
:- begin_module(surface).
:- import(motif).
...
```

If you wish to work with the predicates of the OSF/Motif interface in the top-level dialog (module `user`), you can import the module `motif` as follows:

```
[user] ?- [user].↩
> :- import(motif).↩
> end_of_file.↩
*** consult 'user': loaded in 0.01 sec.

yes
```

Interface to Informix

If the Prolog system has been configured with the Informix interface, an interface to the Informix relational database system is available. For an explanation of how to work with this interface, refer to [4].

If you wish to use the predicates of the Informix interface in modules, you must first import the module `informix` (see also chapter 7).

```
:- begin_module(surface).
:- import(informix).
...
```

If you wish to work with the predicates of the Informix interface in the top-level dialog (module `user`), you can import the module `informix` as follows:

```
[user] ?- [user].↩
> :- import(informix).↩
> end_of_file.↩
*** consult 'user': loaded in 0.01 sec.

yes
```

Constraints package

IF/Prolog system features a constraints package which allows you to organize goal execution more efficiently. The package offers several different constraints classes, which are available as modules.

For information on how to work with constraints and use the built-in predicates, refer to [5].

Chapter 5

The input/output interface

Input and output in IF/Prolog take place using **device drivers** which are managed on the basis of a driver model. A device driver is responsible for operations for a particular stream type, e.g. files, character strings or pipes. Prolog can manage different streams with the same set of generic built-in predicates. Prolog provides the following built-in devices:

- `file` is the device for operating system files
- `null` is the "bit bucket" device
- `pipe` is the device for operating system pipes
- `standard` is the device for the standard streams of the Prolog system
- `string` is the device for input/output using character strings
- `socket` is the device for interprocess communication

In this context, **Stream** is the unique designation of an opened input or output channel, e.g. to a file, terminal or pipe. The designation is an opaque term created by the Prolog system. However, you may also use alias names which you assign to a stream.

This chapter describes both input/output in the top-level dialog and also elementary input/output in Prolog and the main built-in predicates concerned. It then explains how to work with files, pipes and sockets. How to add device drivers to the Prolog system is explained in section 10.3.

5.1 Input/output in the Prolog top-level dialog

Top-level dialog

Input/output in the top-level dialog is handled by the Prolog system using the two **standard streams**. The Prolog system expects queries to be entered on **standard input**, usually the **keyboard**. If the goal is successfully proved, the system outputs the variable instantiations

and, after entry of ↩ , the response `yes` on **standard output**, usually the screen. If the goal cannot be proved, the system outputs the response `no`. In the case of backtracking the Prolog system waits for a request from standard input.

Error messages

Error messages issued by the Prolog system are directed to the **current error output** stream. In trace mode, the Prolog system also writes the trace listing to the current error output. The predicate `set_error/1` can be used to make a stream the current error output stream (see page 82). Any error messages from the Prolog system will then be output to this stream. This can be useful, for example, for detailed study of a trace listing or when a large number of error messages are to be processed, such as when a newly created file is consulted. It also speeds up the time considerably as **standard error output** is character buffered.

Interactive debugger

The various windows and the input and output of the interactive debugger are described in detail in chapter 9.

5.2 Elementary input/output in Prolog

Two groups of input/output predicates are available for input and output: namely, those which refer implicitly to a current stream, and those which require the stream to be explicitly given.

Current streams: There is one current input stream, one current output stream and one current error stream at any given time. At system startup these are the standard streams. If required, you can change this default setting. You should redefine a current stream only if you wish to subsequently use it frequently for input or output. The input/output predicates in this group implicitly address the current streams, since a stream is not explicitly specified.
Input and output using current streams is described in section 5.2.1.

Explicitly specified stream: The specified stream must already exist when the corresponding predicate is called. If the stream is a file, it must be open for reading or writing.
Input and output using explicitly specified streams is described in section 5.2.2.

In Prolog, streams for input and output are referred to by alias names or by means of a system-defined terms which the Prolog system supplies with the predicate `open/3/4`. The

alias names are atoms. The following table provides an overview of the streams which can be addressed. For addressing, either the device name or the alias name can be used. The device name, or driver name, is used only on opening a stream.
The alias name represents a logical reference to a stream and establishes a link between a stream and a name. The name is freely selectable.

Stream	**Alias name**
Standard input	user_input
Standard output	user_output
Standard error output	user_error
Keyboard	keyboard
Screen	screen

The following defaults are set for the three standard streams:

Stream	**Default**
Standard input	Keyboard
Standard output	Screen or window in which Prolog was started
Standard error output	Screen or window in which Prolog was started and error message window

You can also redefine the standard streams when calling Prolog.

Example

The file *goals.pro* is assumed to contain the following goals:

```
Y is 3 + 4.
append(A, B, [a,b,c,d]).
atom(A).
```

If you then call Prolog and specify the file as the input stream, these goals are processed sequentially. The following output appears on the screen:

```
$ prolog < goals.pro

Y       = 7
yes

A       = []
B       = [a,b,c,d]
yes

no

$
```

When the end of the file is reached, Prolog is terminated.

You can use the predicate `stream_copy/2` to change the standard stream assignments during a Prolog session.

5.2.1 Input/output using current streams

Input/output predicates in which no stream is specified address the current streams. The predicate `listing/0/1` also sends its outputs to the current output stream. The Prolog system issues error messages on the current error stream.

When the Prolog system is started, the corresponding standard streams are first opened, i.e.

Stream	**Alias**	**Default**
Current input	current_input	user_input (standard input)
Current output	current_output	user_output (standard output)
Current error output	current_error	user_error (standard error output)

You can refer to the current streams by their alias names in input/output predicates which require a stream to be specified.

You can query or change the settings for the current streams with the following predicates:

Predicate	**Purpose**
current_input(?*Stream*)	Query current input stream
set_input(@*Stream*)	Set current input stream
see(@*Stream*)	Set and if required, open current input stream
seeing(?*Stream*)	Query current input stream
seen	Reset current input stream to `user_input` and if required, close the first stream
current_output(?*Stream*)	Query current output stream
set_output(@*Stream*)	Set current output stream
tell(@*Stream*)	Set and if required, open current output stream
telling(?*Stream*)	Query current output stream
told	Reset current output stream to `user_output` and if required, close the first stream
current_error(?*Stream*)	Query current error stream
set_error(@*Stream*)	Set current error stream
error_tell(@*Stream*)	Set and if required, open current error stream
error_telling(?*Stream*)	Query current error stream
error_told	Reset current error stream to `user_error` and if required, close the first stream

When you declare a file as the current input/output stream, you should first open it with the predicate `open/3/4` and then define it as the current input/output stream with one of

the above predicates. If you wish to declare a pipe as the current input/output stream, you should first create it with `unix_make_pipe/1` and open it with `open/3/4`. Then you define the pipe as the current input/output stream.

With the following predicates, input is using the current input stream and output is using the current output stream.

Predicate	**Purpose**
get_byte(?*ByteCode*)	Read a byte
get_char(?*Char*)	Read a character
get_code(?*CharCode*)	Read a character
get_until(+*SearchChar*, ?*Text*, ?*EndChar*)	Read up to a specified character
nl	Write new line
print(@*Term*)	Write a term
put_byte(+*ByteCode*)	Write a byte
put_char(+*Char*)	Write a character
put_code(+*CharCode*)	Write a character
read(?*Term*)	Read a term
read_term(?*Term*, +*Options*)	Read a term with control options
skip_line	Skip input line
write(@*Term*)	Write a term
write_canonical(@*Term*)	Write a term in normal structure representation and in single quotes
write_formatted(+*Format*, @*TermList*)	Write terms with formatting
write_term(@*Term*, @*Options*)	Write a term using operator notation
writeq(@*Term*)	Write a term in single quotes, using operator notation

5.2.2 Explicitly specifying streams for input/output

Input and output can be performed also from and to an explicitly specified stream. It is not necessary to redefine current streams only for a single input or output operation. Instead, a predicate with an extra argument for the stream can be used.

If you wish to execute the input/output from or to a stream, you must first open this stream prior to the first read or write operation. If the stream is to be a pipe, you should first create it with the predicate `unix_make_pipe(-`*Pipename*`)`. *Pipename* is generated by the Prolog system when the pipe is created.

Open for	**file by calling**	**pipe by calling**
Reading	open(file(*Filename*), read, X)	open(pipe*(Pipename)*, read, Y)
Writing	open(file(*Filename*), write, X)	open(pipe*(Pipename)*, write, Y)
Appending	open(file(*Filename*), append, X)	—

i You can also open a stream (usually a file) by calling `see/1`, `tell/1` or `error_tell/1`.

i File names which contain special or Prolog-specific characters, or which begin with an uppercase letter, must be enclosed in single quotes.

i You can open a stream either for reading or for writing, but not for both simultaneously.

The following predicates require the input/output stream to be specified explicitly. The stream is addressed by means of a name (alias name or term of `open/3/4`). You can also address the standard streams by means of their synonyms.

Predicate	**Purpose**
get_byte(@*Stream*, *?ByteCode*)	Read a byte
get_char(@*Stream*, *?Char*)	Read a character
get_code(@*Stream*, *?CharCode*)	Read a character
get_until(@*Stream*, +*SearchChar*, ?*Text*, ?*EndChar*)	Read up to a specific character
nl(@*Stream*)	Write new line
print(@*Stream*, @*Term*)	Write a term
put_byte(@*Stream*, +*ByteCode*)	Write a byte
put_char(@*Stream*, +*Char*)	Write a character
put_code(@*Stream*, *CharCode*)	Write a character
read(@*Stream*, ?*Term*)	Read a term
read_term(@*Stream*, ?*Term*, @*Options*)	Read a term with control options
skip_line(@*Stream*)	Skip input line
write(@*Stream*, @*Term*)	Write a term
write_canonical(@*Stream*, @*Term*)	Write a term in normal structure representation and in single quotes
write_formatted(@*Stream*, +*Format*, @*TermList*)	Write terms with formatting
write_term(@*Stream*, @*Term*, @*Options*)	Write a term using operator notation
writeq(@*Stream*, @*Term*)	Write a term in single quotes, using operator notation

i Keyboard inputs are normally buffered one line at a time and are passed to the Prolog system when ↩ is pressed. Output to files and pipes is normally also buffered. If required, you can clear the output buffers with `flush_output/0/1`. You can control buffering with `open/3/4`.

You can use the predicate `close/1` to close any stream apart from the standard streams. Should you attempt to close a standard stream with `close/1`, it will have no effect.

The predicate `reset_streams/0` restores the defaults for the standard streams.

5.2.3 Querying information about input/output streams

The following predicates allow you to query which streams are open for input and output, and to request further information about the stream type and device.

Predicate	Purpose
current_error(?*Stream*)	Query current error stream
current_input(?*Stream*)	Query current input stream
current_output(?*Stream*)	Query current output stream
current_device_control(+*Device*, ?*Command*, ?*Arity*)	Query information on device drivers
device_control(+*Device*, @*Command*)	Control device driver
current_stream_control(@*Stream*, ?*Command*, ?*Arity*)	Query information on device drivers
stream_control(@*Stream*, @*Command*)	Control input/output stream
stream_device(@*Stream*, ?*Device*)	Determine device name of an input/output stream
stream_property(?*Stream*, ?*Info*)	Query information on input/output streams
stream_type(@*Stream*, ?*StreamType*)	Query type of an input/output stream
file_test(+*Pathname*, +*AccessMode*)	Check access permissions for a file

Example

Prolog is started with the following redirection of the output:

```
prolog > proout
```

All outputs from the Prolog system are thus written to the file *proout*. Should the file not yet exist, it will be created by the operating system.

The following settings are now in effect for the standard and current streams:

Designation	Opened stream
Standard input	Keyboard
Standard output	File *proout*
Standard error output	Screen
Current input	Keyboard
Current output	File *proout*
Current error output	Screen

The results of executing the following goal are written to the standard output, i.e. to the file *proout*. Note also that since the standard output is redirected, there is no prompt '`[user] ?- `'.

append(A, B, [a,b,c,d]).↩

The variable instantiations are output on standard output. You have to enter ↩ again in order to return to toplevel dialog.
The output resulting from the predicate `write/1` is also written to this file.

write('on current output\n').↩

Error messages such as the following will still appear on the screen, however, because you have not redefined this stream.

asom(a).↩

```
*** E X C E P T I O N: existence_error(procedure,asom / 1)
>>> goal = user : asom(a)
```

You can have outputs made to the screen either by explicitly addressing the screen:

write(screen, 'on the screen\n').↩

```
on the screen
```

or by making the screen the current output stream and then addressing it implicitly:

set_output(screen).↩

The following settings are now effective:

Designation	**Opened stream**
Standard input	Keyboard
Standard output	File *proout*
Standard error output	Screen
Current input	Keyboard
Current output	Screen
Current error output	Screen

You can address the standard output with the name `user_output`.

write('on current output\n').↩

```
on current output
```

write(user_output, 'on standard output\n'). ↩

If you now terminate Prolog, the file *proout* will contain the following:

```
IF/Prolog ...
Copyright (C) Siemens Nixdorf Informationssysteme AG, 1996

A       = []
B       = [a,b,c,d]
yes
on current output
yes
no
yes
yes
yes
on standard output
yes
```

5.3 Working with files

In order to work with Prolog, you must make all the necessary information available in the Prolog database in the form of predicates. When Prolog is called, the system automatically provides a default database (containing the built-in predicates).

Maintaining and modifying data in the database alone would not only be laborious but also rather inefficient since all the data would be lost at the end of the Prolog session. You should therefore work with files. You can

- conveniently write and modify files using the editor.
- retain files as Prolog sources independently of Prolog sessions.
  When a Prolog session is terminated, the entire database is erased from main memory. Anything you wish to keep must be stored in files.
- compile Prolog source files.
  Prolog modules which you wish to compile must be held in a file.
- structure large programs in files.
  Files allow you to give your programs a clear structure. For example, all predicates which logically belong together can be written to one file.

5.3.1 Editing files with an editor

Files are most conveniently created and manipulated with an **editor**, provided by most operating systems. An editor is a programm designed to support creating and manipulating

(text) files. An explanation of the use and facilities of editors is beyond the scope of this manual.

Various editors are available under the UNIX operating system as tools for processing. The `vi` editor belongs to the basic configuration of a UNIX system and is thus almost certainly installed on your system. Under other operating systems, other text processing tools are available.

Calling an editor from the operating system

You call the desired editor from the operating system shell. Loading, editing and writing to a file are done in the usual manner.

Calling an editor in a Prolog session

You can call the editor during a Prolog session with:

`edit(`*Filename*`)`	edit the file *Filename*
`edit`	edit the last file processed with `edit/0/1` during the Prolog session

Filename must be a syntactically valid file name for the operating system. The Prolog system searches in the order below:

- first for a file called *Filename*
- then, if this cannot be found, for a file called *Filename.pro*

If the search is unsuccessful, a file named *Filename.pro* is created automatically. The suffix *.pro* is added only if it is not already included in *Filename*.

The predicate `edit/0` or `edit/1` now handles the following automatically:

- temporary exit from Prolog
- call to the editor with the specified file name as a parameter
- return to Prolog
- after user confirmation, reading in the updated file again with `reconsult/1`.

You can thus modify files containing Prolog source text during a Prolog session without quitting Prolog completely. The changes take effect immediately, provided you have saved them with the editor and then reconsulted the file. You can also edit files containing no Prolog clauses but any input In this case you should not let Prolog reconsult the file. Otherwise, it can have unintended results when it is read into the database.

Changing the editor

The default setting for the predicates `edit/0/1` is operating system dependent. You can change this default setting with `default_editor/2`; for example:

```
[user] ?- default_editor(_, emacs).↩
```

This selects the editor `emacs`. From now on, `edit/0/1` will cause `emacs` to be invoked.

Possible errors

Three types of error or exception can occur:

- Exceptions when the call is made, such as
  - *Filename* is not a valid operating system file name
  - the file cannot be created or modified because the user does not have the proper access permissions.
- Errors during editing, such as
  - typing errors
  - incorrect use of editor commands

  (see relevant editor description).
- Errors and exceptions when the modified file is reconsulted, such as
  - the file does not exist because you exited the editor with a command which did not save the text in a new file.
  - the file contains invalid Prolog clauses. These clauses are not included in the database when `reconsult/1` is called. The Prolog system issues appropriate syntax error messages.
  - the file contains clauses associated with a predicate which cannot be changed (see `modify_mode/3`). These clauses are not included in the database. The Prolog system generates an appropriate error message.

i Deleting a predicate from the file does not automatically delete it from the database (see the description of `reconsult/1` in the *IF/Prolog Reference Manual* [1]). `retract/1/2` or `abolish/1/2` remove only non-compiled predicates from the database.

5.3.2 Reading a file into the database

You can read a file into the database either when Prolog is first called or during a Prolog session. In each case a distinction is made according to whether the file contains Prolog source text or is a compiled file.

In the case of Prolog source text, the procedure is called **consulting**; with compiled files it is called **loading**.

Reading in a file when calling Prolog

When you initiate Prolog, there are various ways of reading a file into the database:

- The option `-st` *startfile* allows you to read in a special start file; `-st -` causes this operation to be suppressed. If no `-st` option is specified, a default start file (if present) is read in (see page 66).
- You can also read a file into the database by specifying the option `-c` and/or `-l` (see chapter 4.1.1).

File contents	Option
Prolog source text	-c *filename*
Compiled Prolog code	-l *filename*

Reading in a file during a Prolog session

File contents	Predicate
Prolog source text	`consult`(*Filename*)
	[*File1, File2, ...*]
	`reconsult`(*Filename*)
	[*-File1, -File2 ...*]
Compiled Prolog code	`load`(*Filename*)

For a description of the various predicates, refer to the *IF/Prolog Reference Manual* [1].

i The association of the predicates to the original file is not stored in the database.

i You can remove only a module as a complete entity from the database (see chapter 7.5.3: *Removing a module from the database*). Individual clauses or predicates can be deleted with `retract/1/2` or `abolish/1/2`.

5.3.3 Writing the database to a file

You can store clauses read into the database by a user during a Prolog session with the predicates `listing/0/1`. Note that `listing/0/1` cause the output to be written on the current output stream (see page 83). If the output is to be directed to a file, the current output stream must be first redefined accordingly (see page 85).

Example

```
[user] ?- open(file(clausefile), write, Z), set_output(Z), listing,
set_output(screen), close(Z).↩

yes
```

The predicate `listing/1` allows you to selectively write clauses with a specific functor (and arity).

5.4 Process management and communication

The Prolog system offers a set of predicates for process management and communication with signals, pipes and sockets. This section provides a tabular overview of these predicates and an example of process communication using pipes and sockets. The predicates are described in detail in the *IF/Prolog Reference Manual* [1]. There you will also find further short examples.

5.4.1 Predicates for process management

The Prolog system provides the following predicates for process management:

Predicate	Purpose
system	Start the operating system command interpreter
system(+*Command*)	Execute an operating system command
system(+*Command*, ?*ExitStatus*)	Execute an operating system command and query the exit status
system(+*Command*, ?*Input*, ?*Output*)	Execute an operating system command, with input and output streams specified
system(+*Command*, ?*Input*, ?*Output*, ?*Error*, ?*Pid*)	Execute an operating system command, with input, output and error streams specified
chdir(+*Dir*)	Change current directory
getcwd(?*Dir*)	Query current working directory
exec(+*Command*)	Overlay the process and execute a command
exec(+*Prog*, @*Args*)	Call a program and overlay the current process
unix_fork(?*Pid*)	Spawn process
unix_getpid(?*Pid*)	Query process ID of current process
unix_wait(?*Pid*)	Wait for termination of a child process
unix_wait(?*Pid*, ?*ExitStatus*)	Wait for termination of a child process and query the exit status

5.4.2 Predicates for process communication with signals

The Prolog system provides the following predicates for process communication with signals:

Predicate	Purpose
current_signal(?*Signal*, ?*Mode*)	Query signals and modes
signal_control(+*Signal*, +*Command*)	Control signal processing
signal_handler(+*Goal*, ?*Signal*, +*Handler*)	Define local signal handling
unix_kill(+*Pid*, +*Signal*)	Send signal to a process

5.4.3 Predicates for process communication using pipes

The Prolog system provides the following predicates for process communication using pipes:

Predicate	Purpose
stream_copy(@*Stream1*, @*Stream2*)	Redefine streams
unix_make_pipe(-*Pipename*)	Create a pipe

Example for process management and communication using pipes

The following predicate `activate/2` first creates two pipes and then spawns a child process. In the child process, these two pipes are used as the standard input and standard output streams. Now the child is overlaid by a program *Prog* with the arguments *Args*. This program reads an integer from standard input, calculates the square of the integer and writes the result to standard output. The parent process uses the predicate `task/2` to communicate with the child process.

```
/*********************************************************/
/* Create pipes and generate child process               */
/* The child process is overlaid by the program Server   */
/* with the parameters Args.                             */
/*********************************************************/

activate(Server,Args):-
        unix_make_pipe(Send_Pipe),
        unix_make_pipe(Receive_Pipe),
        unix_fork(Pid),
        father_son(Pid,Send_Pipe,Receive_Pipe,Server,Args).

/*********************************************************/
/* The parent process opens the pipes                    */
/* The child process opens the pipes and makes them the  */
/* standard input and standard output and is finally     */
/* overlaid by the program Server                        */
/*********************************************************/

father_son(0,Send_Pipe,Receive_Pipe,Server,Args):-
        !,
        open(Send_Pipe, read, Send),
        stream_copy(Send, user_input),
        open(Receive_Pipe, write, Receive),
        stream_copy(Receive, user_output),
        exec(Server, Args).

father_son(_Pid,Send_Pipe,Receive_Pipe,_,_):-
        open(Send_Pipe, write, Send),
        open(Receive_Pipe, read, Receive),
        task(Send, Receive).

/*********************************************************/
/* task/2 requests an input, reads it from the current   */
/* input stream and passes it to evaluate/3.             */
/* This continues until end_of_file is entered.          */
/*********************************************************/
```

```
skip_header(Stream) :-
        repeat,
        get_until(Stream,'\n',Line,_),
        Line=ready,!.

task(Send,Receive):-
        skip_header(Receive),
        repeat,
        write('>'),
        get_until('\n',Command,Stop),
        evaluate(Command,Send,Receive),
        Stop==end_of_file,!,
        stop_server(Send,Receive).

/*****************************************************************/
/* stop_server/2 closes the pipes.                               */
/* evaluate/3 passes the input to the Send_Pipe and writes the    */
/* result from the Receive_Pipe to the current output stream      */
/*****************************************************************/

stop_server(Send,Receive) :-
        close(Send),
        close(Receive),
        write(quit),nl.

evaluate('',_,_) :- !.

evaluate(Command,Send,Receive) :-
        write(Send,Command),nl(Send),flush_output(Send),
        get_until(Receive,'\n',Result,_),
        write(Result),nl.
```

The program *Server*, which overlays the child process, may be implemented in any language. It could, for example, be the following Prolog program (file *server.pro*):

```
program:-
        write('ready\n'),flush_output,
        repeat,
        get_until('\n',Command,Stop),
        evaluate(Command),
        Stop==end_of_file.

evaluate(Command):-
        atom_chars(Command,CharList),
        exception_handler(number_chars(Number,CharList),
```

```
                error(syntax_error('illegal number'), _),
                fail),
        integer(Number),!,
        calculate_square(Number).

evaluate(Command) :-
        write(Command),write('?'),nl,flush_output.

calculate_square(Number):-
        Limit is sqrt(maxint),
        Number =< Limit,!,
        Square is Number * Number,
        write(Square),nl,flush_output.

calculate_square(_) :-
        write('overflow !'),nl,flush_output.
```

A dialog such as the following could take place:

```
[user] ?- activate(prolog, ['-c', 'server.pro']).↩



>45
2025
>56
3136
>5678
32239684
>99999
overflow !
>stop
stop?
> quit
```

5.4.4 Predicates for process communication using sockets

The Prolog system provides the following predicates for process communication using sockets:

Predicate	Purpose
current_socket(?*Domain*, ?*Type*, ?*Socket*)	Query communication sockets
get_socket_option(@*Socket*, ?*Option*, ?*Value*)	Get socket option
net_service(+*Service*, ?*Protocol*, ?*Port*)	Query communication service
select(@*Streams*, +*Timeout*, -*ReadyStreams*)	Synchronous input multiplexing
set_socket_option(@*Socket*, +*Option*, +*Value*)	Set socket option
socket(+*Domain*, +*Type*, -*Socket*)	Create communication socket
socket_accept(@*Socket*, ?*Addr*, -*NewSocket*)	Accept a connection
socket_bind(@*Socket*, ?*Addr*)	Bind a name to a socket
socket_connect(@*Socket*, @*Addr*)	Connect a socket
socket_listen(@*Socket*)	Listen for connections
socket_listen(@*Socket*, +*Quelen*)	Listen for connections
socket_receive(@*Socket*, ?*List*)	Receive data from socket
socket_receive(@*Socket*, ?*Addr*, ?*List*)	Receive data from socket
socket_send(@*Socket*, @*List*)	Send data to socket
socket_send(@*Socket*, @*Addr*, @*List*)	Send data to socket
socket_shutdown(@*Socket*)	Close a socket
socket_shutdown(@*Socket*, +*How*)	Close a socket

A **socket** is the basic object for interprocess communication. It is an endpoint of communication, represented in Prolog by an opaque term.

The **type** of a socket specifies the communication properties. The following socket types are supported:

stream sockets provide bidirectional, reliable, sequenced and unduplicated flow of message data.

datagram sockets support bidirectional data flow, but don't guarantee that the message is sequenced, reliable or unduplicated.

The **domain** of a socket specifies the address family. The following socket domains are supported:

stream domain address is a combination of an Internet address and port (delivery slot). The Internet address can be given symbolically as a host name or explicitly as an address in dotted notation. The ports are managed by the system.

unix domain address is a unique filename, used only for rendezvous. The corresponding file is created by the server process and used by client process(es).

The processes wishing to communicate with each other create sockets with the predicate `socket/3`. Messages can be sent only after the communication has been established. The communicating processes are classified conceptually into **server(s)** and **client(s)**. A server is basically a process providing services for other processes, and a client is a process utilizing these services.

Communication between connected processes

In order to establish reliable communication between processes, **stream** sockets are used and the processes are connected. This is semantically comparable with communication using pipes.

The server process must first **bind** the socket to an address before being able to provide the service (predicate `socket_bind/2`). The server can bind the socket to a specific port or let the system choose the port. It then **listens** for incoming communication requests from clients (predicates `socket_listen/1/2`). A client process must know which kind of service it needs and at which address the service is provided. It uses the same address (filename or port) as the server for establishing a connection (predicate `socket_connect/2`). The server accepts the incoming requests (predicate `socket_accept/3`) and the data transfer can begin (predicates `socket_send/2` and `socket_receive/2`). Once the communication has been completed, the processes can be disconnected by closing the sockets (predicate `socket_shutdown/1/2`).

Communication between processes without connection

Connectionless interactions are typically realized using **datagram** sockets. Processes are still conceptually divided into a server and client(s). There is no need for the server to bind a socket to an address, neither for the client to connect to this address. Instead, each message sent includes a destination address that can be queried by the receiving process (predicates `socket_send/3` and `socket_receive/3`).

Generic stream input/output using sockets

After a connection has been established, a socket is both readable and writable. Prolog streams, however, are either readable or writable, but not both. Therefore, to be able to use the generic stream input/output predicates, two streams must be opened.

Input/output multiplexing

Several clients can be simultaneously connected to one server. In order to be able to handle incoming messages, input/output multiplexing can be used (predicate `select/3`).

5.4.5 Predicates for computer network information

The Prolog system provides the following predicates for accessing information on computer networks:

Predicate	Purpose
current_host(?*Host*)	Query names of current host
host_addr(+*Host*, ?*Addr*)	Query host address
host_addr(-*Host*, +*Addr*)	Query host name
net_service(+*Service*, ?*Protocol*, ?*Port*)	Query communication service

The information provided is extracted either from Network Information Service (NIS), Domain Name Service (DNS) or local operating system databases. More information can be found in the operating system documentation in sections dealing with networking and communication.

Example for client communication using sockets

The following Prolog program shows simple client requesting the **time** service from a server (this is a standard service on networked computers).

! The Prolog system must be configured with Multiple Precision Integer Support for correct arithmetics (without overflow).

```
gettime(Host) :-
        gettime(Host, GMT1970),
        localtime(GMT1970, Year, Month, Day, _DoW, _DoY, Hrs, Min, Sec),
        write_formatted('%02d.%02d.%4d %02d:%02d:%02d',
                [ Day, Month, Year, Hrs, Min, Sec ]).

gettime(Host, GMT1970) :-
        net_service(time, tcp, TimePort),       % Well known number: 37
        host_addr(Host, _),                     % only to check Host
        socket(internet, stream, Socket),
        socket_connect(Socket, Host : TimePort),
        open(Socket, read, Stream),
        select([Stream], off, _),               % wait for data
        socket_receive(Socket, TimeData),
        close(Stream),
        socket_shutdown(Socket),

        TimeData = [ B1, B2, B3, B4 ],
        GMT1900 is B1 * 256 * 256 * 256 + B2 * 256 * 256 + B3 * 256 + B4,
        Magic is 0x83AA7E80,
        GMT1970 is GMT1900 - Magic.
```

Example for client/server communication using sockets

The following Prolog text implements a simple server reading Prolog queries from a socket and writing all answers to a socket. It also shows how to use sockets with generic in-

put/output predicates.

```
/*
?-      server.
(Port number will be written, use the same number for client!)
*/

server :-
        socket(internet, stream, Socket),
        socket_bind(Socket, Port),
        write(port = Port), nl,
        socket_listen(Socket),
        socket_accept(Socket, Host:Port2, NewSocket),
        write(accepted(Host:Port2)), nl,

        open(NewSocket, read, Input),
        open(NewSocket, write, Output),
        repeat,
                read(Input, Message),
                write('Server received: '),
                write(Message), nl,
        not handle_server(Message, Output),
        !,
        close(Input),
        close(Output),
        socket_shutdown(Socket),
        socket_shutdown(NewSocket).

handle_server(end_of_file, _) :- !, fail.
handle_server(halt, _) :- !, fail.
handle_server(help, Output) :-
        !,
        write_canonical(Output, 'Enter Prolog queries. End with halt.'),
        write(Output, '. '),
        flush_output(Output).
handle_server(Query, Output) :-
        exception_handler(findall(Query, Query, Answer),
                Exception, Answer = Exception),
        writeq(Output, Answer),
        write(Output, '. '),
        flush_output(Output).
```

The following Prolog text implements a client, which connects to the above server, reads queries from the user and sends them to the server.

```
/*
```

```
?-      client(Host, Port).     % Port number written by server
% Server outputs: accepted(Internet-Number, Port)

Client now waits for queries, sends them to the server,
which answers etc. End with 'halt.'
*/

client(Host, Port) :-
        open_client(Host, Port, Socket),
        work_client(Socket).

open_client(Host, Port, Socket) :-
        integer(Port), !,
        socket(internet, stream, Socket),
        socket_connect(Socket, Host : Port).

work_client(Socket) :-
        open(Socket, read, Input),
        open(Socket, write, Output),
        repeat,
                write('Client: '), flush_output,
                read(Query),
        not handle_client(Query, Input, Output),
        close(Input),
        close(Output),
        socket_shutdown(Socket).

handle_client(end_of_file, _, _) :- !, fail.
handle_client(halt, _Input, Output) :-
        write(Output, halt),
        write(Output, '. '),
        flush_output(Output), !, fail.
handle_client(Query, Input, Output) :-
        writeq(Output, Query),
        write(Output, '. '),
        flush_output(Output),
        read(Input, Answer),
        write('Server answered: '),
        write(Answer), nl,
        !,
        Answer \= end_of_file.
```

Chapter 6

The Prolog grammar rules

This section deals with the syntax of the Prolog grammar rules. At the same time, it explains how the Prolog system converts these rules into Prolog clauses during consultation and how these clauses can then be used for the syntactical analysis of sentences in the corresponding language.

Finally, the complete syntax of the production rules is defined by reference to the associated grammar.

6.1 Context-free grammar

An important step in the automatic processing of natural or formal languages consists in breaking down the input stream of words or characters into sentences in the relevant language and then analyzing those sentences syntactically.

This can be done for each language on the basis of a grammar which defines:

- The set of words (**tokens**) in the language.
  These are identified in the grammar by means of terminal symbols (**terminals**).
- The various categories of token sequences, i.e. phrases which occur when sentences are formed and which are part of the input stream.
  These are indicated by non-terminal symbols (**non-terminals**).
- The internal structure of phrases. A set of rules called **production rules** is provided for this purpose.
- A **start symbol** from the set of non-terminals.

Whereas a grammar of this sort must be constructed inductively for **natural** languages (this is usually only partly successful), a **formal** language (command language, programming language) is often a set of sentences generated deductively with a predefined grammar.

A grammar will be particularly amenable to automatic processing if all of its production rules have the following structure:

$$\begin{array}{llll} S_0 & \text{may consist of} & S_1 & \text{followed by} \quad S_2 \\ & & & \text{followed by} \quad S_3 \\ & & & \vdots \\ & & & \text{followed by} \quad S_i \\ & & & \vdots \\ & & & \text{followed by} \quad S_n. \end{array}$$

Or, expressed concisely in operator notation:

$S_0 \longrightarrow S_1 \ , \ S_2 \ , \ S_3 \ , \ \ldots \ , \ S_i \ , \ \ldots \ , \ S_n. \ (n \geq i \geq 1)$

In this case the start symbol S_0 is a non-terminal symbol, and the body elements S_1 , ... , S_n are non-terminal or terminal symbols.

This type of grammar is known as a **context-free grammar**.

In a context-free grammar the start symbol is used to generate a sentence in the following way (deriving the start symbol):

- Begin with the start symbol.
- Replace each non-terminal S by the right-hand side of a production rule $S \longrightarrow \ldots$
- Repeat this until a sequence of terminal symbols is left over. This sequence is then a sentence in the language.

The set of sentences derived in this manner is called the **language** generated by the grammar.

Example

A grammar consisting of

```
Terminals:      [the], [dog], [barks], [runs]

Non-terminals:  sentence, nominal_phrase, verbal_phrase,
                article, noun, verb

Production rules:

        sentence         -->  nominal_phrase, verbal_phrase.
        verbal_phrase    -->  verb.
        nominal_phrase   -->  article, noun.
        article          -->  [the].
        noun             -->  [dog].
```

```
          verb            -->  [runs].
          verb            -->  [barks].

    Start symbol:      sentence
```

generates the language consisting of the two sentences:

```
[the , dog , runs]

[the , dog , barks]
```

Here, consecutive lists were combined to form a single list.

6.2 Simple grammar rules

Prolog enables you to enter production rules directly in infix operator notation. To create a grammar, we shall begin with the rules on which the above example was based, namely:

- Terminals:
  Prolog uses single-element lists as terminal symbols. The elements in these lists are the tokens of the processed token sequence.

- Non-terminals:
  Prolog atoms.

- Production rules, made up of

  - a head consisting of a non-terminal symbol,
  - a production operator `-->`, followed by
  - the body of the production rule. This consists of at least one or a series of terminal or non-terminal symbols, separated by ",". The end of a production rule is indicated by a period "." .

- Start symbol:
  A start symbol does not have to be specially identified as such. For derivation purposes, any non-terminal can be used as a start symbol.

Extensions to these rules are outlined in sections *Additional arguments* (page 107) and *Explicit Prolog goals* (page 110).

6.2.1 Non-terminals in a production rule

As already mentioned, production rules of this sort are automatically compiled into syntactically correct clauses by the Prolog system when reading text with `consult/1`, `reconsult/1` or `compile/1`. Each non-terminal symbol is converted into a structure with an arity of 2. A grammar rule

s_0 `-->` s_1 `,` s_2 `,` s_3 `,` ... `,` s_n`.`

with non-terminals s_0`,` ... `,` s_n is compiled to form the clause

$s_0(R_0, R_n)$ `:-` $s_1(R_0, R_1), s_2(R_1, R_2), \ldots, s_n(R_{n-1}, R_n)$.

Example

```
sentence -->
        nominal_phrase ,
        verbal_phrase.
```

is compiled to form the clause

```
sentence(S_Input,S_Tail) :-
        nominal_phrase(S_Input,N_Tail),
        verbal_phrase(N_Tail,S_Tail).
```

(The variable names in the database have been replaced by more meaningful names in the text).

The two arguments in the structures $s_0(R_0, R_n)$ `,` ... `,` $s_n(R_{n-1}, R_n)$ each form a **difference list**: the first argument is a list, and the second argument is a list tail of the first list. The elements in the difference list are obtained by subtracting the two lists, i.e. by removing the elements of the list tail from the end of the first list.

Example

The difference lists with the pairs

```
List    =    [the,dog,barks]
Tail    =                 [ ]
```

or

```
List    =    [the,dog,barks,the]
Tail    =                   [the]
```

or

```
List    =    [the,dog,barks,dog,dog,dog]
Tail    =                   [dog,dog,dog]
```

are all equivalent to the list

```
Diff    =    [the,dog,barks]
```

On the other hand, the list pair

```
L1 = [the,dog,barks]
L2 = [runs]
```

does not form a difference list.

The first list in a difference list is frequently referred to as an **input list**.

6.2.2 Terminals in a production rule

Unlike the non-terminal symbols of a production rule, the terminal symbols do not have equivalents in the associated subgoals of the compiled rule. They should correspond directly to tokens from the input list and are therefore taken into account in both parts of the difference list which implement the processing of this input list.

Example

The grammar from the first example is extended by adding the production rule:

```
sentence -->
        nominal_phrase , [and] , verbal_phrase.
```

After consulting, the database contains the clause

```
sentence(SE,SR) :-
        nominal_phrase(SE,S0), S0 = [and|S1R], verbal_phrase(S1R,SR).
```

The format `[ and |` *S1R* `]` of the list tail implies that it must begin with the terminal symbol `[and]` and that the remaining tail, unified with *S1R*, is available for further syntactical analysis.

In general, a terminal symbol may be located

- at any position in the body of the production rule
- after the non-terminal symbol in the head of the production rule

Terminal symbols at the beginning of the production rule body are inserted by the compiler at the beginning of the input list. In all other cases, the terminal symbols are added at the beginning of the list tail of the preceding non-terminal symbol.

Example

In the grammar database of the first example, the production rule

```
verb --> [barks].
```

has the corresponding clause

```
verb( In, Out) :-
        In = [barks|Out].
```

Example

A grammatical rule in the form

```
phrase1, [word] --> phrase2, phrase3.
```

recurs in the database as the clause

```
phrase1( In1, [word|Out3]) :-
        phrase2( In1, Out2),
        phrase3( Out2, Out3).
```

As soon as `phrase2` and `phrase3` are recognized in the input stream *In1*, the terminal `word` is added to the front of the list *Out3* of the remaining terminals. `word` is then the first terminal symbol which is available for further syntactical analysis.

6.2.3 Deriving a start symbol

Once all the production rules of a grammar have been compiled and read into the database, you may form the derivations of a start symbol `start` by calling the Prolog goal `start(`*Head*`,[])`. *Head* is instantiated successively on all lists of tokens corresponding to the syntactical structure of the non-terminal `start`.

Example

Using the grammar from the first example, the following result is output:

```
[user] ?- sentence(Head, []).↩

Head = [ the , dog , runs ] ;↩
Head = [ the , dog , barks ] ;↩

no
```

6.2.4 Syntactic analysis

If conversely a list of tokens *InputList* is given then, by calling the goal `start(`*InputList,Tail*) you may check whether there is a token sequence at the beginning of *InputList* which has the syntactic structure of the phrase `start`. If there is, the goal succeeds and *Tail* is unified with that part of *InputList* which no longer belongs to this token sequence. A similar process takes place internally when the subgoals belonging to this goal are proved. The list tail remaining in a subgoal then forms the input list for the following subgoal or in the case of the final subgoal, it forms the list tail for the entire goal.

Example

The Prolog system has compiled the grammar given in the first example and read it into the database. We now ask

```
[user] ?- sentence([ the, dog, barks, the, dog ], S_Tail).↩

S_Tail  = [the, dog] ↩

yes
```

The answer shows that, in the input list, the sublist `[the, dog, barks]` is made up of the phrase `sentence` and the tail `[the, dog]` is left over. When this goal is proved, the first thing that happens is that the subgoal `nominal_phrase([the, dog, barks, the, dog],`*N_Tail*) is activated. Since `[the, dog]` is a valid `nominal_phrase`, the goal succeeds and *N_Tail* is unified with `[barks, the, dog]`. This forms the input list for the second subgoal `verbal_phrase([barks, the, dog],`*S_Tail*). Thus for the entire goal, *S_Tail* = `[the, dog]` remains as the list tail.

6.3 Additional arguments

In addition to the above, Prolog structures of any arity are permitted for non-terminal symbols in a rule of grammar. This does not, however apply to lists since these are already used for the terminal symbols. During compilation, the explicitly specified arguments in a structure of this type are incorporated unchanged as arguments in the corresponding subgoal of the Prolog clause. Behind these arguments the compiler inserts two further variables or lists which, as described are intended for the difference list for the purpose of handling the input list.

The production rule

$s_0(T0_1, \ldots, T0_{i0}) --> s_1(T1_1, \ldots, T1_{i1}), \ldots, s_n(Tn_1, \ldots, Tn_{in}).$

is thus compiled into

$s_0(T0_1, \ldots, T0_{i0}, R_0, R_n) :- s_1(T1_1, \ldots, T1_{i1}, R_0, R_1), \ldots, s_n(Tn_1, \ldots, Tn_{in}, R_{n-1}, R_n).$

Example

One example of how to use these additional arguments is the construction of a syntax tree in parallel to the syntactical analysis. A grammar extended in this manner not only confirms whether an input phrase is a valid sentence in a language but also generates additional information on its syntactic structure.

The rules of our original grammar are now extended for each non-terminal symbol by an argument containing the associated node in the syntax tree.

```
sentence(sentence(N_ph,V_ph))  -->  nominal_phrase(N_ph),
                                    verbal_phrase(V_ph).
verbal_phrase(v_ph(V))         -->  verb(V).
nominal_phrase(n_ph(A,N))      -->  article(A), noun(N).
article(article([the]))        -->  [the].
noun(noun([dog]))              -->  [dog].
noun(noun([bird]))             -->  [bird].
verb(verb([barks]))            -->  [barks].
verb(verb([runs]))             -->  [runs].
```

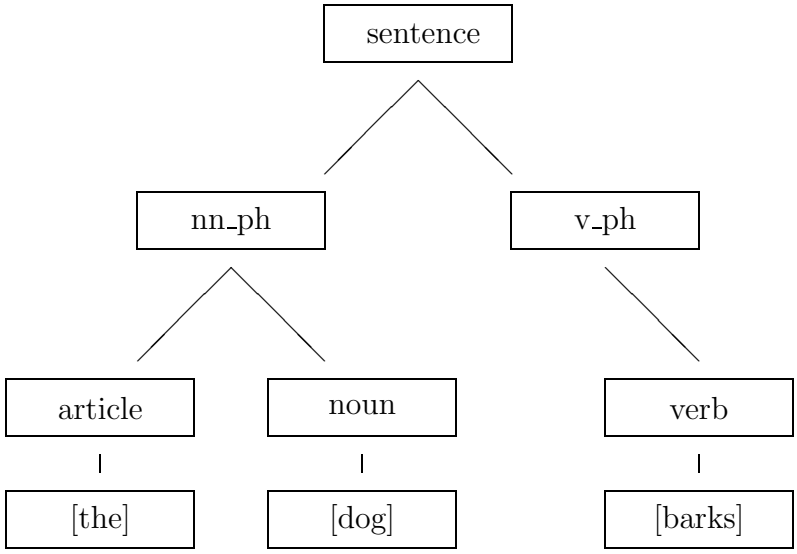
The database now has clauses of the form

```
verbal_phrase(v_ph(V),Inp,Tail) :- verb(V,Inp,Tail).
```

and the Prolog system responds to the query

```
[user] ?- sentence(Syntaxtree, Sentence, []).↩


Syntaxtree = sentence(n_ph(article([the]),noun([dog])),
                v_ph(verb([barks])))
Sentence   = [the,dog,barks] ;

Syntaxtree = sentence(n_ph(article([the]),noun([dog])),
                v_ph(verb([barks])))
Sentence   = [the,dog,runs] ;

Syntaxtree = sentence(n_ph(article([the]),noun([bird])),
                v_ph(verb([barks])))
Sentence   = [the,bird,barks] ;

Syntaxtree = sentence(n_ph(article([the]),noun([bird])),
                v_ph(verb([runs])))
Sentence   = [the,bird,runs] ;

no
```

The first solution generated the sentence `[the, dog, barks]` and the following syntax tree:



Admittedly however, this grammar also recognizes the token sequence `[the, bird, barks]` as a correct sentence and rightly so, since a syntactical analysis cannot provide any information on meaning, i.e. on the semantic content.

Example

An option line of the form

```
p1 +      p2 -      p3 -
```

is to be read and checked for syntactical correctness, and a list [+,-,-] of the input parameter values generated. If a given option is not specified, the associated default value is to be assumed.
`default1`, `default2` and `default3` supplement options missing from the input stream.

```
input( [P1,P2,P3] ) -->
        blanks, default1, parameter1(P1),
        blanks, default2, parameter2(P2),
        blanks, default3, parameter3(P3),
        blanks.

blanks --> [' '], blanks.
blanks --> [].

parameter1(P1) -->
        [p,'1'], blanks, par(P1).
parameter2(P2) -->
```

```
        [p,'2'], blanks, par(P2).
parameter3(P3) -->
        [p,'3'], blanks, par(P3).

default1, [p,'1',+] --> [].
default1-->[].
default2, [p,'2',+] --> [].
default2-->[].
default3, [p,'3',-] --> [].
default3-->[].

par(+) --> [+].
par(-) --> [-].
```

If no option is entered, the default values are set:

```
[user] ?- input(List, [], []).↩

List    = [+,+,-]
```

Parameter 1 and parameter 3 are set explicitly and the blanks are skipped:

```
[user] ?- atom_chars( ' p1 - p3+', Name), input( List, Name, []).↩

List    = [-,+,+]
```

6.4 Explicit Prolog goals

Prolog also allows explicit Prolog goals as body elements in a grammar rule. In order for these elements to be recognizable as such by the compiler, they must be enclosed in braces {}. They will then be inserted unchanged at the corresponding position in a resultant Prolog clause.
One exception to this is the **cut**. A call of the predicate `!/0` can be inserted without braces as a body element in a grammar rule.

Since the variable names which describe the input sequence and its tail in the difference lists are assigned by the Prolog system itself, they cannot be accessed directly for explicit Prolog goals. However, if appropriate information is required, it can be made available by means of additional arguments. By inserting explicit goals and using additional arguments, a context-free grammar can be turned into an **attributed grammar**. This not only performs syntactic analysis but may also contain semantic elements.

Example

The sample grammar given above will accept meaningless sentences. This can be avoided by replacing the production rule for `sentence` by

```
sentence(sentence(N_ph,V_ph)) -->
        nominal_phrase(N_ph),
        verbal_phrase(V_ph),
        {
                not (N_ph = n_ph(X,noun([bird])),
                V_ph = v_ph(verb([barks])))
        }.
```

In a more comprehensive grammar, however, we would like find a more systematic way to exclude semantically incorrect sentences.

Example

Explicit Prolog goals are used here to calculate the numeric values of verbally specified integers in addition to performing a syntactical analysis. The restriction to three-digit integers can easily be removed by means of additional grammar rules.

```
number(0)    -->  [zero].
number(N)    -->  digit(H), [hundred],
                  and_tens(T), {N is H * 100 + T}.
number(N)    -->  sub_tens(N).

and_tens(0)  --> [].
and_tens(N)  --> [and], sub_tens(N).

sub_tens(N)  --> digit(N).
sub_tens(N)  --> teen(N).
sub_tens(N)  --> tens(T), and_digit(D), {N is T + D}.

and_digit(0) --> [].
and_digit(N) --> digit(N).

digit(1) --> [one].      teen(10) --> [ten].
digit(2) --> [two].      teen(11) --> [eleven].
digit(3) --> [three].    teen(12) --> [twelve].
digit(4) --> [four].     teen(13) --> [thirteen].
digit(5) --> [five].     teen(14) --> [fourteen].
digit(6) --> [six].      teen(15) --> [fifteen].
digit(7) --> [seven].    teen(16) --> [sixteen].
```

```
digit(8) --> [eight].     teen(17) --> [seventeen].
digit(9) --> [nine].      teen(18) --> [eighteen].
                          teen(19) --> [nineteen].

tens(20) --> [twenty].
tens(30) --> [thirty].
tens(40) --> [forty].
tens(50) --> [fifty].
tens(60) --> [sixty].
tens(70) --> [seventy].
tens(80) --> [eighty].
tens(90) --> [ninety].
```

The associated program now responds to queries of the following type:

```
[user] ?- number(Value, [ two, hundred, and, twenty, three ], []).↩

Value   = 223 ↩

yes
```

Besides performing this syntactical analysis and calculating the value, the program is also capable of providing an appropriate verbal expression for a given value (synthesis).

```
[user] ?- number(101, X, []).↩

X       = [one,hundred,and,one]
```

6.5 Syntax of the production rules

The most important options offered by the syntax of the grammar rules are explained above. Below you will find the entire syntax of this form of grammar described with these very same production rules. Then we will deal with one of these rules which has not yet been discussed.

```
grammar --> g_rule.                                    %1
grammar --> g_rule, grammar.                           %2

g_rule --> g_rule_head, ['-->'] , g_rule_body , ['.']. %3

g_rule_head --> non_terminal.                          %4
g_rule_head --> non_terminal , [','] , terminal.       %5
```

```
g_rule_body --> body_element.                          %6
g_rule_body --> g_rule_body , [','] , g_rule_body.     %7

g_rule_body --> [','],['('] ,body_element, [',']
         ,body_element, [')'].                         %8
g_rule_body --> g_rule_body , [';'] , g_rule_body.     %9

g_rule_body --> [';'],['('] ,body_element, [',']
         ,body_element, [')'].                         %10

body_element --> ['!'].                                %11

body_element --> ['{'] , prolog_goal , ['}'].          %12

body_element --> ['('] , g_rule_body , [')'].          %13

body_element --> non_terminal.                         %14
body_element --> terminal.                             %15

terminal --> ['[]'].                                   %16

terminal --> ['['] , token_sequence , [']'].           %17

token_sequence --> token.                              %18
token_sequence --> token , [','] , token_sequence.     %19
```

Here, `token` stands for the smallest syntactical unit of the language described by the grammar. Any Prolog structure is permitted as a `token`, generally atoms.
`non_terminal` stands for a particular category of token sequences. `non_terminal` may be Prolog atoms, Prolog normal structures and expressions, but not lists.

Rules %8 and %10 imply that, instead of the operator notation given in %7 or %9 for conjunction or disjunction, it is also possible to use normal-structure notation with the functors ',' and ';'.
The bracketing option described in rule %13 is employed with operators ',' and ';' to influence the precedence of the terms accordingly. In this case, the same rules apply as with normal Prolog terms (see *IF/Prolog Reference Manual* [1], chapter on syntax). As with Prolog clauses, disjunction should be avoided wherever possible for the sake of clarity.

Example

```
sentence--> ( sub_sentence ; sub_sentence , ['and'] , sentence).
```

This grammar rule is compiled into the following Prolog clause which is added to the database:

```
sentence(A,B) :-
                sub_sentence(A,B)
        ;
                sub_sentence(A,C),
                C = [and|D],
                sentence(D,B) .
```

If the rule is split up as follows:

```
sentence --> sub_sentence.
sentence --> sub_sentence , ['and'] , sentence.
```

the clauses are more easily read:

```
sentence(A,B) :-
        sub_sentence(A,B) .
sentence(A,B) :-
        sub_sentence(A,C),
        C = [and|D],
        sentence(D,B) .
```

Chapter 7

The Prolog module concept

IF/Prolog features a powerful module concept designed to provide you with support for creating large program packages and program libraries.

In a module, a set of predicates and clauses are combined to form a logical unit. The predicates are visible only within the module, but you can also make one or more predicates visible for other modules (**export**). Modules thus allow you to conceal implementation details (information hiding) and avoid naming conflicts. The predicates which you have exported can be made visible in other modules (**import**). The set of exported module objects therefore constitutes the interface to other modules.

The built-in predicates of IF/Prolog are also stored in modules. When Prolog is called, the modules of the standard system are visible. The extensions to the Prolog system, e.g. for the OSF/Motif interface or the constraint package, are stored in separate modules which you must import. Once you have imported a module, all the predicates exported there are visible in your own module. You may not redefine the modules of the Prolog system. The built-in predicates are stored in the following modules:

Module	**Meaning**
system	Contains the built-in predicates of the standard system
system...	Further system modules
user	Contains user-defined predicates which are not assigned to any other module
motif	Contains the predicates of the OSF/Motif interface
informix	Contains the predicates of the Informix interface
const_delay const_linear const_domain	Contain the predicates of the constraints package

The predicate `current_module/1` allows you to query all modules which have been loaded or imported into the database.

Example

```
[user] ?- current_module(M), write(M), nl, fail.↩


system
user
system.debug.breakpoint
system.debug.command
system.debug.alpha
system.debug.surface_sh
system.debug.com_sh
system.debug.surface_handler
system.debug.configuration
system.debug.error
system.debug.global
system.debug.kernel
system.debug
system.break
informix
motif
const_linear
const_domain
const_delay

no
```

As soon as you call the Prolog system, the built-in module `user` becomes the current module. All predicates which you enter on the standard input loop `break/0` are proved in the context of this module (see also chapter 8). The predicate `current_default_module/1` allows you to query the current module and `set_default_module/1` allows you to define a module of your own as the current module.

The following predicates are available to you when programming with modules:

Predicate	**Purpose**
Module : Predicate	Specify the module of the called predicate
Predicate @ Module	Specify module context for predicate call
calling_context(?Module)	Query the module argument of meta-predicates
current_default_module(?Module)	Query current module
current_module(?Module)	Query modules
save_module(+Module)	Store current module in a file
set_default_module(+Module)	Set current module
unload(+Module)	Remove module from the database

This chapter describes the syntax and structure of modules and objects which you can manage in modules. You will learn how to import and export predicates, compile a module and process the module in the database.

7.1 Modules

A module comprises two parts: the module **interface** and the module **body**, which may be both logically and physically separate. The module interface and module body are interlinked by their common module name.

The module concept is static, i.e. the module interface is not created until the module is loaded. You can dynamically modify only the module body.

7.1.1 Directives

Modules are defined by syntactic constructs known as **directives**. A directive is a single-arity structure and has the functor `:-` (where `:-` is a built-in operator). The directives are evaluated by the following built-in predicates which are used for loading Prolog programs.

```
consult/1
reconsult/1
compile/1
./2
```

Outside the context of these predicates, directives have no effect. The following table provides an overview of the directives available.

Directive	Effect
:- begin_module +*Module*	Define the body of a module
:- char_conversion(@*CharacterIn*, @*CharacterOut*)	Define a character conversion
:- discontiguous @*PredicateList*	Declare discontiguous predicate definition
:- dynamic @*PredicateList*	Declare predicates as modifiable
:- end_module	Conclude the definition of a module
:- end_module *Module*	Conclude the definition of a module
:- ensure_loaded +*Filename*	Insert a Prolog file in the source text once
:- export @*PredicateList*	Export predicates
:- import +*Module*	Import exported predicates of a module
:- import +*PredicateList* from +*Module*	Import certain exported predicates of a module
:- include +*FileName*	Include a Prolog file in the source text
:- init +*Goal*	Initialization goal
:- initialization +*Goal*	Initialization goal
:- meta @*PredicateList*	Declare metapredicates
:- module +*Module*	Define a module
:- multifile @*PredicateList*	Declare predicates as being scattered over several files
:- op(+*Priority*, +*Assoc*, @*Names*)	Define and delete operators
:- private @*PredicateList*	Declare predicates as private
:- reexport +*Module*	Reexport predicates
:- reexport @*PredicateList* from +*Module*	Reexport certain predicates

Any module-specific declaration is performed using directives.

7.1.2 Logical and physical files

IF/Prolog makes a distinction between physical and logical files. A logical file may comprise a number of physical files. You use the `include` +*Filename* or the `ensure_loaded` +*Filename* directive to incorporate physical files into logical files. When Prolog texts are consulted, Prolog replaces these directives with the contents of the physical file *Filename*.

Filename is an atom which represents the path and the name of a file in accordance with the operating system used. If *Filename* is not an absolute pathname, the Prolog system interprets the pathname relative to the current file. For example, if the file */usr/joey/prolog/test.pro* contains the directive `:- include('../example/frame.pro')`, on loading the `include` directive is replaced by the file */usr/joey/example/frame.pro*.

If the file cannot be opened, or if *Filename* is not an absolute pathname, Prolog searches for the file in the directories defined under the Prolog flag `search_path` (see predicate `current_prolog_flag/2`).

Example

physical file: *append.pro*

```
append([], L, L).
append([X|L1], L2, [X|L3]) :- append(L1, L2, L3).
```

physical file: *member.pro*

```
:- include 'append.pro'.
member(X, [X|L]).
member(X, [_|L]) :- member(X, L).
```

physical file: *list.pro*

```
is_list([]).
is_list([_|L]) :- is_list(L).
:- include 'member.pro'.
```

When you consult the physical file *list.pro*, the physical files *member.pro* and *append.pro* will be read in, too. Your database will then contain the following logical file:

```
is_list([]).
is_list([_|L]) :- is_list(L).
append([], L, L).
append([X|L1], L2, [X|L3]) :- append(L1, L2, L3).
member(X, [X|L]).
member(X, [_|L]) :- member(X, L).
```

7.2 Module objects

IF/Prolog manages two classes of objects in modules:

- predicates
- global variables

Objects of these types are known as module objects. Each module object is assigned to precisely one module, known as the original module. There are no module objects which are assigned globally (i.e. assigned to all modules or to none at all). This applies also to all built-in module objects.

By default, module objects are visible only in their original module. This means that it is possible to define objects with the same name in different modules without name conflicts.

You can declare predicates publicly (export). You can then make them visible in other modules (import). The indicator (name and arity) of a predicate is significant for its identification.

Each module object in a module must be uniquely identified. This means that it is impossible to make predicates (with the same indicator) from several modules visible in any one module, or to define a predicate as private and at the same time import it.

Built-in system predicates are made visible in all modules as standard, provided they are not overlaid by local definitions or by explicitly imported predicates with the same name and the same arity.

Global variables are only visible in their original module. The module system has no language constructs for making them visible in other modules. They can, however, be accessed by using an explicit module argument for the corresponding built-in predicates. Only the name of a global variable is significant in its identification.

7.2.1 Predicates

IF/Prolog manages both user-defined predicates and built-in system predicates in modules. Initially, all predicates are visible only in their original module. This applies both to predicates defined in Prolog and to C predicates. You can access C predicates only while the body of their original module exists.

Built-in predicates automatically become visible when the module body is defined, provided you do not redefine them in the module interface and thereby declare them as private.

The following directives declare predicates as private:

```
:- export PredicateList .
:- meta PredicateList .
:- dynamic PredicateList .
:- private PredicateList .
```

If you declare an imported predicate as private in the module interface using one of the above directives, you **overlay** the predicate i.e. the imported predicate becomes invisible again.

i If you have reexported a predicate in the module interface, you may not overlay it with a private predicate in the module body.

7.2.2 Metapredicates

In contrast to all other predicates, metapredicates have an additional argument, the module argument, or meta-argument. The module argument is handled separately by Prolog. If the module argument is not specified during goal execution for a metapredicate, it is implicitly assigned the name of the module in which the predicate was called. This enables metapredicates to determine the name of the calling module. You can use metapredicates to perform certain actions within the calling module. This is particularly desirable if a predicate has Prolog goals as arguments which are to be executed in the calling module.

A metapredicate can access the module argument using the built-in predicate `calling_context/1`:

`calling_context( ?`*Module* `)`

Module is unified with the implicitly or explicitly specified module name of the metapredicate call. In Prolog, built-in predicates are referred to as metapredicates if they have goals as arguments, allow the database to be modified, or manage global variables.

Metapredicates which modify the database

```
'.'/2                    abolish/1                asserta/1/2
asserta_with_names/2/3   assertz/1/2              assertz_with_names/2/3
consult/1                edit/0/1                 reconsult/1
retract/1/2              retract_with_names/2/3
```

Metapredicates which inspect the database

```
clause/2                 clause_with_names/3      current_predicate/1
current_visible/2        debug_mode/3             list_mode/3
listing/0/1              modify_mode/3            predicate_mode/3
predicate_type/2         save_module/1
```

Metapredicates which have goals as parameters

```
bagof/3                  catch/3                  context/2
debug/1                  exception_handler/3      findall/3
once/1                   setof/3                  signal_handler/3
trace/1
```

Metapredicates which execute user predicates in the calling module

```
print/1/2                write_term/2/3
```

Metapredicates which manage global variables

```
current_global/1        get_global/2            pop_global/2
push_global/2           set_global/2            unset_global/1
```

The language construct `call/1` has an important role to play in implementing metapredicates. It has a module argument which is handled by Prolog in the same way as in metapredicates. If the module argument is not set explicitly, it is assigned the name of the module in which the language construct is used.

As its argument, `call/1` has a goal which can be statically or even dynamically known. The module argument determines the module in which this goal is to be executed.

Example

This example shows the implementation of a small debugger.

```
:- module test_help.
:- meta [(#)/1].

:- begin_module test_help.
:- op(900, fx, #).

(# Goal) :-
        calling_context(Module),
        writeq(call(Goal)), nl,
        call(Goal) @ Module,
        writeq(exit(Goal)), nl.

(# Goal) :-
        writeq(fail(Goal)), nl,
        fail.

:- end_module test_help.
```

You can trace goal by placing the operator `#` in front of the goal. `#` is defined as an operator at load time using the `op` directive. It can thus also be used by other modules.

```
:- import test_help.

predicate :-
        goal1,
        # goal2,
        # goal3,
        ...
        goaln.
```

The predicate `#/1` has been declared as a metapredicate, so its module argument is implicitly set to the name of the calling module. `#/1` uses this to execute *Goal* in the calling module.

i It is possible to simulate a predicate call from any module by explicitly specifying the module argument when using metapredicates. This is important particularly in the implementation of metainterpreters if they are to operate on programs spread over a number of modules.

Explicit qualification of predicate activators

Irrespective of its visibility, any predicate can be called system-wide by means of explicit qualification. A predicate is explicitly associated with a module in which it is visible using the visibility operator (`:/2`). You can use the visibility operator as follows:

```
Module : Activator
```

It is not necessary for the *Predicate*, called by *Activator*, to have been exported from the specified *Module* or for the *Module* to have been imported into the current module. You only need to know the module in which the predicate you wish to qualify is visible.
If you use the visibility operator, both the *Module* and the *Predicate* must be instantiated at load time. However, the predicate need not be defined at load time.

i You can explicitly qualify predicates only; you may not qualify language constructs.

Implicit qualification of predicates

When a term is transformed into a goal all the predicates not explicitly qualified are implicitly qualified, i.e. identified with their original module. The only exceptions to this transformation are those predicates which are parenthesized by `call/1` explicitly or implicitly (if the goal is a variable).

A term is transformed into a goal whenever it is loaded into the Prolog database (with `consult/1, assertz/1/2`, etc.) or when a term parenthesized by `call/1` has to be made available for execution.

Transformation is subject to the following rules:

- If a Prolog goal is a variable, the variable is parenthesized by the control structure `call/1`, and the module argument of `call/1` is set to the name of the current module.
- If a Prolog goal is an explicitly qualified metapredicate without an explicit module argument, the module argument is set to the name of the current module.
- If a Prolog goal is an unqualified predicate, it is qualified with the name of the module from which it was imported. If the predicate was not imported, it is qualified with the name of the current module.

- If a Prolog goal is an unqualified visible predicate whose module argument is explicitly set although it is not defined as a metapredicate, the module argument is ignored.

- If a Prolog goal is a metapredicate whose module argument is not explicitly set, the module argument is set to the name of the current module. The same applies to the language construct `call/1`.

- If a Prolog goal is a language construct (except for `call/1`) which itself has subgoals as arguments, the above rules are applied to the subgoals.

Example

```
:-module expert.
:-meta [minimize/2].

:-module timetable.
:-export [connection/3, duration/3].

:-module plan.
:-begin_module plan.
:-import expert.
:-import timetable.

journey(From,To,Holidays):-
        minimize(Price,connection(From,To,Price)),
        duration(From,To,Days),
        Days =< Holidays.
:- end_module.
```

The body of rules `journey/3` is transformed as follows:

```
journey(From,To,Holidays) :-
        expert:minimize(Price,connection(From,To,Price)) @ plan,
        timetable:duration(From,To,Days),
        system:(Days =< Holidays).
```

Language constructs

The following indicators denote language constructs and can therefore not be used as indicators for user predicates.

Indicator	Language construct
, / 2	Conjunction of goals
; / 2	Disjunction of goals
-> / 2	Conditional execution of goals
\+/1	Negation of a goal
not / 1	Negation of a goal
call / 1	Indirect activation of goals
true / 0	Always true
fail / 0	Always false
! / 0	Cut
: / 2	Explicit qualification
@ / 2	Setting of the module argument
:- / 1	Directive
:- / 2	Rule
--> / 2	Grammar rule

7.2.3 Global variables

Global variables are managed only in modules. They are visible only within their original module. You can, however, access global variables in other modules.
All predicates which access global variables are implemented as metapredicates.

Predicates for accessing global variables

Predicate	Purpose
current_global(?*Name*)	Query global variables
get_global(+*Name*, ?*Value*)	Query the value of a global variable
pop_global(+*Name*, ?*Value*)	Remove value of a global variable
push_global(+*Name*, @*Value*)	Set value of a global variable
set_global(+*Name*, @*Value*)	Set value of a global variable
unset_global(+*Name*)	Delete a global variable

If you explicitly specify the meta-argument, these predicates are executed in the context of the specified module. In this way you can access the global variables of these modules.

Example

The following example shows how you can modify global variables in other modules.

```
:- module generator.
```

```
:- export [reset/0, gensym/1].

:- begin_module generator.

reset:- set_global(index,0).

gensym(Symbol):-
        get_global(index,Index),
        NewIndex is Index+1,
        set_global(index,NewIndex),
        concat_atom(['symbol_',Index],Symbol).

:- init set_global(index,0).

:- module objects.

:- begin_module objects.
:- import generator.

create_object(Name):- gensym(Name).

reset(N):- set_global(index,N) @ generator.
```

7.3 Module interface

A module consists of two parts: the module interface and the module body which may be both physically and logically separate. The link between the two parts is established by their common module name.

The module interface and the module body can be terminated by means of an optional `end_module` directive. The `end_module/1` directive must contain the name of the current module as its argument.

The module interface is introduced by the `module` directive and constitutes the public part of a module. Predicates defined in the module body can be exported, and thus imported into other modules.

`:- module` *Module*.

$$\left[\left\{\begin{array}{l} :-\texttt{export}\ \mathit{PredicateList}. \\ :-\texttt{meta}\ \mathit{PredicateList}. \\ :-\texttt{reexport}\ \mathit{Module}. \\ :-\texttt{reexport}\ \mathit{PredicateList}\ \texttt{from}\ \mathit{OtherModule}. \end{array}\right\}\right]$$

$$\left[\left\{\begin{array}{l} :-\texttt{end\_module}. \\ :-\texttt{end\_module}\ \mathit{Module}. \end{array}\right\}\right]$$

Module is an atom which specifies the name of the module. The name must be unique system-wide, i.e. there must not be more than one module with the same name. If you have already defined a module with the name *Module*, it is redefined, and all module objects and declarations which belonged to the previous module are deleted. Redefining the module interface automatically deletes an existing module body. You cannot redefine system modules.

The `module` directive creates a new module. The module interface ends with the following `module`, `begin_module` or `end_module` directive, or at the end of the logical file.

A logical file may contain more than one module interface. A module interface, however, must be completely defined within a single logical file.

i If you have implemented C predicates for a module, they cannot be accessed until after the module body has been defined. If a module body is deleted, the associated C predicates remain hidden until the body has been redefined.

7.3.1 Exporting predicates

The `export` directive is used in the module interface. It has the argument *PredicateList* which corresponds to a list of predicate indicators. The predicates in the list are made publicly visible; thus they can be made visible in other modules by means of the `import` directive.

All predicates which you list in an `export` directive are implicitly declared as private. This prevents the local definition of a predicate being overlaid by an `import` directive in the module body. In this way you can redefine built-in system predicates within a module.

If you have implemented predicates with the C interface in a module, you can make them visible with the `export` directive. But you must ensure that the predicate definitions themselves are not made available until after the module body has been defined.

i Predicates which you have implemented in C and have not exported are visible only in their original module.

Example

```
:-module permutation.
:-export [permute/2].

:-module order.
:-export [ordered/1].

:-module sort.
:-export [sort/2].
:-begin_module sort.
```

```
:-import permutation.
:-import order.

sort(List,SortedList):-
        permute(List,SortedList),
        ordered(SortedList).
```

The `sort` module exports the `sort/2` predicate and declares it public. Initially, it remains visible only in the `sort` module. Exported predicates from other modules are made visible with the `import` directive. Thus, in the above example the `import` directives make the following predicates visible in the `sort` module:

- `permute/2` (from the original module `permutation`)
- `ordered/1` (from the original module `order`)

7.3.2 Exporting metapredicates

Using the `meta` directive you can export metapredicates in the module interface and define them in the declaration part of the module body. This section describes the semantics of the `meta` directive in the module interface.

The `meta` directive has the argument *PredicateList*, which corresponds to a list of predicate indicators. The predicates in the list are declared publicly as metapredicates; thus you can make them visible in other modules using the `import` directive.

You can export only metapredicates with the `meta` directive. You must use the `export` directive to export all other predicates.

All the predicates which you have made public with the `meta` directive in the interface are implicitly declared as private. This prevents an `import` directive in the module body from overlaying the private definition of a predicate.

i You may not export metapredicates with the `export` directive.

Example

This example defines the module `output`. It exports the predicate `portray/1` and the metapredicate `output/1`. `output/1` will output a term in the context of the calling module.

If a predicate with the indicator `portray/1` is visible in the calling module, it will be called in order to output the term. If `portray/1` cannot be successfully executed, the term will be output with `writeq/1` in the context of the calling module.

```
:-module output.
:-export [portray/1].
:-meta [output/1].

:-begin_module output.
output(Term) :-
        calling_context(Module),
        call(portray(Term)) @ Module, !.
output(Term) :-
        writeq(Term).

portray(_):-fail.

:-end_module output.
```

In the following the module `persons` is defined; this module exports the predicate `collects/2`. In addition, a predicate `portray/1` is defined which outputs the structure of `collects/2` in a special way.

```
:-module persons.
:-export [collects/2].

:-begin_module persons.

collects(bill, [stamps,pictures]).
collects(jane, [butterflies]).

portray(collects(Person, [])):-
        write_formatted('%q collects nothing.\n', [Person]).

portray(collects(Person, Things):-
        write_formatted('%q collects ', [Person]),
        show_collection(Things).

show_collection([Thing]):-
        !,
        write_formatted('%q .\n', [Thing]).

show_collection([Thing|Tail]):-
        write_formatted('%q and ', [Thing]),
        show_collection(Tail).

:-end_module persons.
```

The body of the module `operation` is also defined. It imports the modules `output` and

`persons`, making the predicates `output/1`, `portray/1` and `collects/2` visible. The definition of `portray/1` from the `persons` module is not visible.

```
:- module operation.
:- begin_module operation.
:- import output.
:- import persons.

program :-
        collects(Person, Things),
        output(collects(Person, Things)),
        output(collects(Person, Things)) @ persons.

:- end_module operation.
```

Calling the predicate `program/0` generates the following output:

```
        collects(bill, [stamps, pictures])
        bill collects stamps and pictures .
```

This output is the result of the following individual steps:

For `output(`*Term*`)`:

Current module	**Goal**
operation	output(Term) @ operation
output	call(portray(Term)) @ operation
operation	portray(Term)
output	fail
output	writeq(Term) @ operation

For `output(`*Term*`) @ persons`:

Current module	**Goal**
operation	output(Term) @ persons
output	call(portray(Term)) @ persons
persons	portray(Term)
	write_formatted('%q collects ', [bill]) @ persons
	show_collection(Things)
	...

7.3.3 Reexporting all public predicates of a module

The directive `reexport(`*Module*`)` makes all public predicates of the module *Module* available to other modules. The `reexport` directive requires that the module interface of *Module* be

known. *Module* is an atom which defines the name of the module whose module interface is to be reexported.

You can reexport only predicates which you have already exported in the module interface of *Module*.

i You must not overlay reexported predicates with private predicates. You must not reexport predicates which have the same indicator but have been declared in different modules.

Example

In this example the module `utility` exports all predicates which are exported in the modules `permute`, `order` and `sort_new`.

```
:- module permutation.
:- export [permute/2].
:- end_module permutation.

:- module order.
:- export [ordered/2].
:- end_module order.

:- module sort_new.
:- export [sort/2].
:- end_module sort_new.

:- module utility.
:- reexport permutation.
:- reexport order.
:- reexport sort_new.
:- end_module utility.
```

7.3.4 Reexporting specific public predicates of a module

The selective directive `reexport` *PredicateList* `from` *Module* makes a subset of the public predicates of the module *Module* available to other modules. The `reexport` directive requires that the module interface of *Module* be known. *Module* is an atom which defines the name of the module whose module interface is to be reexported. *PredicateList* is a list of predicate indicators. The predicates contained in the list are to be reexported. You must have exported all predicates of the *PredicateList* in the module interface of *Module*.

You can reexport only predicates which you have already exported in the module interface of *Module*.

i You must not overlay reexported predicates with private predicates. You must not reexport predicates which have the same indicator but have been declared in different modules.

7.3.5 Changes to the module interface

Changes to the module interface are ignored by other modules that already exist. In other words, the redefinition of a module interface does not affect the visibility of predicates in other modules.

For reasons of program transparency, any changes to a module interface should be made only during the module development or debugging phase.

7.4 The module body

The module body is introduced by the `begin_module` directive and constitutes the private part of a module. It contains the implementation details of a module. The module body is divided syntactically into two parts: the **declaration** part and the **implementation** part. The module body ends with the next `module, begin_module` or `end_module` directive, or at the end of the logical file.

:- `begin_module` *Module*.
Declaration part
Implementation part

$$\left[\left\{\begin{array}{l} :-\texttt{end\_module}. \\ :-\texttt{end\_module}\ \mathit{Module}. \end{array}\right\}\right]$$

$$\text{Declaration part::=}\left[\left\{\begin{array}{l} :-\texttt{import}\ \mathit{Module}. \\ :-\texttt{import}\ \mathit{PredicateList}\texttt{from}\mathit{Module}. \\ :-\texttt{discontiguous}\ \mathit{PredicateList}. \\ :-\texttt{multifile}\ \mathit{PredicateList}. \\ :-\texttt{private}\ \mathit{PredicateList}. \\ :-\texttt{dynamic}\ \mathit{PredicateList}. \\ :-\texttt{meta}\ \mathit{PredicateList}. \end{array}\right\}\right]$$

$$\text{Implementation part::=}\left[\left\{\begin{array}{l} \mathit{Literal}. \\ \mathit{Literal} :- \mathit{Goal}. \\ \mathit{GrammarRule}. \\ :-\texttt{initialization}\ \mathit{Goal}. \\ :-\texttt{init}\ \mathit{Goal}. \end{array}\right\}\right]$$

Module is an atom which specifies the name of the module whose body is to be defined. For a module body to be defined, the associated module interface must exist, i.e. you must always load the module interface of a module before its body. If the body of a module has already been declared, all the module objects and private declarations of the module

are deleted before the body is reloaded. The module interface itself is not affected by the `begin_module` directive.

The `begin_module` directive implicitly causes all the system predicates from the `system` module, and all the built-in operators to be made visible. If in the module interface, you have declared a predicate whose indicator is identical to a system predicate, the corresponding system predicate is not made visible. These predicates can be overlaid with a local definition.

i You cannot redefine the body of a system module. The prefix `system...` is also reserved for system module names and should not be used for module names (see also page 115).

Declaration part

The declaration part of the module body contains only directives which make predicates from different modules visible, define operators or declare private (i.e. local) predicates and their properties.

The individual directives and their application are described in detail below. Declarations in the module body must not contradict the module interface in any way. For example, if you have declared predicates as public with an `export` directive, you may not declare them locally as metapredicates.

Implementation part

The implementation part of the module contains the predicate definitions. In addition, you can define goals there which will be executed in its module context once the module has been loaded.

You can define any number of module bodies in a logical file; the declaration part must be completely defined within a single logical file. The implementation part of a module body may be spread over any number of logical files. This enables it to be partially loaded or redefined with `consult/1` or `reconsult/1`.

7.4.1 Importing all public predicates of a module

The `import` directive is used in the declaration part of the module body. It enables you to import all the predicates which were exported in *Module*. The `import` directive requires that the module interface of *Module* be known.

Module is an atom which defines the name of the module whose module interface is to be imported.

If you declare a predicate as private with an `export, meta, dynamic` or `private` directive, it is not imported. If you have already imported a predicate from a different module, it is not imported again by the general `import` directive. In this case, a warning appears.

i In this respect the general `import` directive differs from the selective `import` directive. With the selective `import` directive, predicates which have already been imported can be overlaid. The general `import` directive, on the other hand, imports only predicates with indicators which have been neither declared as local nor imported from a different module.

Example

The module `circular_note` imports the interface of the modules `letterbox` and `persons`. This makes the exported predicates visible in the module `circular_note`.

```
:- module letterbox.
:- export [send/2].
:- end_module letterbox.

:- module persons.
:- export [person/1].
:- end_module persons.

:- module circular_note.
:- export [circular/1].

:- begin_module circular_note.
:- import letterbox.
:- import persons.

circular(Message) :-
        person(Person),
        send(Person, Message),
        fail.
circular(_).

:- end_module circular_note.
```

7.4.2 Importing specific predicates of a module

The selective directive `import` *PredicateList* `from` *Module* allows you to make visible a subset of the exported predicates of the module *Module*. The `import` directive requires that the module interface of the module *Module* be known.

Module is an atom which defines the name of the module from which the predicates are to be imported. *PredicateList* is a list of predicate indicators. The predicates in the list are to be made visible. You should have made all the predicates in *PredicateList* publicly

visible in the module interface of the module *Module* by means of `export`, `reexport` or `meta` directives.

If you have declared a predicate as private with an `export, meta, dynamic` or `private` directive or reexported a predicate, you can no longer import it from a different module. Any attempt to import it will not be treated as an error; instead, the system will issue a warning.

If you have already imported a predicate from a different module, it will be imported again. This means that previous `import` declarations may be partially overlaid by subsequent ones. However, the Prolog system issues a warning to draw your attention to the redeclaration of a predicate.

i In this respect the general `import` directive differs from the selective `import` directive. With the selective `import` directive, predicates which have already been imported can be overlaid. The general `import` directive, on the other hand, imports only predicates with indicators which have been neither declared as local nor imported from a different module.

Example

In this example, two module interfaces are defined. the predicate `sort/2` is exported in both interfaces. In the module body of the module `operation`, the module `quicksort` is first imported. Consequently, the built-in standard predicate `sort/2` is implicitly overlaid by the definition from the module `quicksort`. The predicate `permute/2` is then imported from the module `utilities`.

As a result, all the built-in system predicates are visible in the `operation` module, with the exception of `sort/2`. In addition, the `sort/2` predicate from the `quicksort` module and the `permute/2` predicate from the `utilities` module are visible.

If there are any private predicates defined in the module, then these will of course also be visible within the module.

```
:- module utilities.
:- export [sort/2, permute/2].
:- end_module utilities.

:- module quicksort.
:- export [sort/2].
:- end_module quicksort.

:- module operation.

:- begin_module operation.
:- import quicksort.
:- import utilities, [permute/2].
:- end_module operation.
```

7.4.3 Declaring predicates as distributed

You can split up the clauses of a predicate with the directives `discontiguous/1` and `multifile/1`. If the definition of a predicate is split without using an appropriate directive, the Prolog system issues a warning.

By using the `discontiguous/1` directive you can distribute the definition (clauses) of a predicate within a file.

The `multifile/1` directive allows (the clauses of) a predicate to be defined in several files.

Both directives have the argument *PredicateList*, which corresponds to a list of predicate indicators.

7.4.4 Declaring predicates as private

The `private` directive is used in the declaration part of the module body. It has the argument *PredicateList*, which corresponds to a list of predicate indicators. The predicates in the list are declared as private.

The private declaration of a predicate prevents the private definition of a predicate from being overlaid by an `import` directive. The `private` directive thus also allows you to redefine built-in system predicates. The directives `export, dynamic` and `meta` implicitly declare a predicate as private.

If you reexport a predicate in the module interface, then you may not declare it as private.

Example

In this example, the system predicates `assertz/1` and `retract/1` are redefined locally. This enables important actions of this module to be logged without having to change the actual implementation of the module.

The system predicates `assertz/1` and `retract/1` in the module `database` are made invisible by the private declaration. You can, however, continue to call them by means of an explicit qualification.

```
:- module database.
:- export [db_assert/1, db_retract/1, db_access/1].
:- end_module database.

:- begin_module database.
:- dynamic [database/1].
:- private [assertz/1, retract/1].

assertz(Tuple) :-
```

```
        writeq(insert(Tuple)), nl,
        system: assertz(Tuple).

retract(Tuple) :-
        system: retract(Tuple),
        writeq(delete(Tuple)), nl.

db_assert(Tuple) :-
        assertz(database(Tuple)).
db_retract(Tuple) :-
        retract(database(Tuple)).
db_access(Tuple) :-
        database(Tuple).

:- end_module database.
```

7.4.5 Declaring predicates as dynamic

The `dynamic` directive is used in the declaration part of the module body. It has the argument *PredicateList*, which corresponds to a list of predicate indicators. The predicates in the list are declared as dynamic.

At runtime, you can inspect (`clause/2, listing/0/1`, ...) and modify (`retract/1/2`, ...) dynamic predicates. Activation of dynamic predicates which do not have any clauses does not trigger the exception `existence_error(procedure)`, instead the predicate is evaluated with `fail`.

All predicates listed in a `dynamic` directive are also implicitly declared as private. This prevents the private definition of a predicate from being overlaid by an `import` directive. This also allows you to redefine built-in system predicates within a module.

If you wish to reexport a predicate in the module interface, then you should not declare it as dynamic using the `dynamic` directive.

i All the predicates defined in the module body which are not declared as dynamic are by default static (unless you specify the system parameter `-debug` when invoking Prolog). You can neither inspect nor modify static predicates, but may delete them with `abolish/1`.
You cannot declare predicates which you have implemented with the C interface as dynamic.

Example

This example defines the module `storage_administration`. It exports the predicates `receipt/2, outlet/2` and `stock/2`. The predicate `stock/2` shows the current stock level; since it is subject to change, this predicate is declared as dynamic.

```
:- module storage_administration.
:- export [receipt/2, outlet/2, stock/2].

:- begin_module storage_administration.
:- dynamic [stock/2].

stock(screws, 50).
stock(nails, 100).
stock(fillets, 20).

receipt(Product, Quantity) :-
        retract(stock(Product, So_far)),
        !,
        NewQuantity is So_far + Quantity,
        assertz(stock(Product, NewQuantity)).

receipt(Product, Quantity) :-
        assertz(stock(Product, Quantity)).

outlet(Product, Quantity) :-
        stock(Product, So_far),
        So_far >= Quantity,
        NewQuantity is So_far - Quantity,
        retract(stock(Product, So_far)),
        assertz(stock(Product, NewQuantity)).

:- end_module storage_administration.
```

i If you export a dynamic predicate, it cannot be modified in a different module. You may modify a dynamic predicate in the original module only. This restriction is lifted by explicitly specifying the original module of a dynamic predicate:
`retract(stock(Product, Quantity)) @ storage_administration`
The following call of `assertz/1` in a different module would not affect the module `storage_administration`:
`assertz(stock(P,Q)).`

7.4.6 Declaring metapredicates

The `meta` directive allows you to export metapredicates in the module interface and to declare metapredicates in the declaration part of a module body. This section describes the semantics of the `meta` directive in the declaration part of the module body.

The `meta` directive has the argument *PredicateList*, corresponding to a list of predicate indicators. Each predicate in the list is declared as a metapredicate. Each predicate is also

implicitly declared as private. This prevents the private definition of a predicate from being overlaid by an `import` directive.

Local definition of metapredicates is seldom required since a local metapredicate is usually called only within the module. If the module argument is implicitly assigned by the module system, it will, by default, bear the name of its own module.

i Each metapredicate must be declared before it can be defined. You may not declare a predicate which you have exported or reexported locally as a metapredicate.

7.4.7 Defining operators

The `op` directive allows you to declare and define operators in the module body and module interface. The `op` directive differs from the predicate `op/3` only in as much as the directive is evaluated at load time and thus takes effect immediately.
The operators which you define with the predicate or directive have global validity.

How to define and delete operators is described in the section on *Syntax* in the *IF/Prolog Reference Manual* [1].

Example

In the module `expression`, expressions of the type *Result* `:=` *Expression* are evaluated as follows. First all subexpressions are evaluated and then a predicate is called whose arguments are the evaluated subexpressions and a variable for the result. The name of the predicate is determined by the name of the function.

```
:- module expression.
:- meta [(:=)/2].

:- begin_module expression.
:- op(700, xfx, (:=)).

Value := Expression :-
        number(Expression), !, Value = Expression.

Value := Expression :-
        functor(Expression, Name, Arity),
        Expression =.. [_|Args],
        evaluate(Args, Params),
        calling_context(Module),
        evaluate_function(Name, Arity, Params, Value, Module).

evaluate_function(Name, Arity, Params, Result, _) :-
```

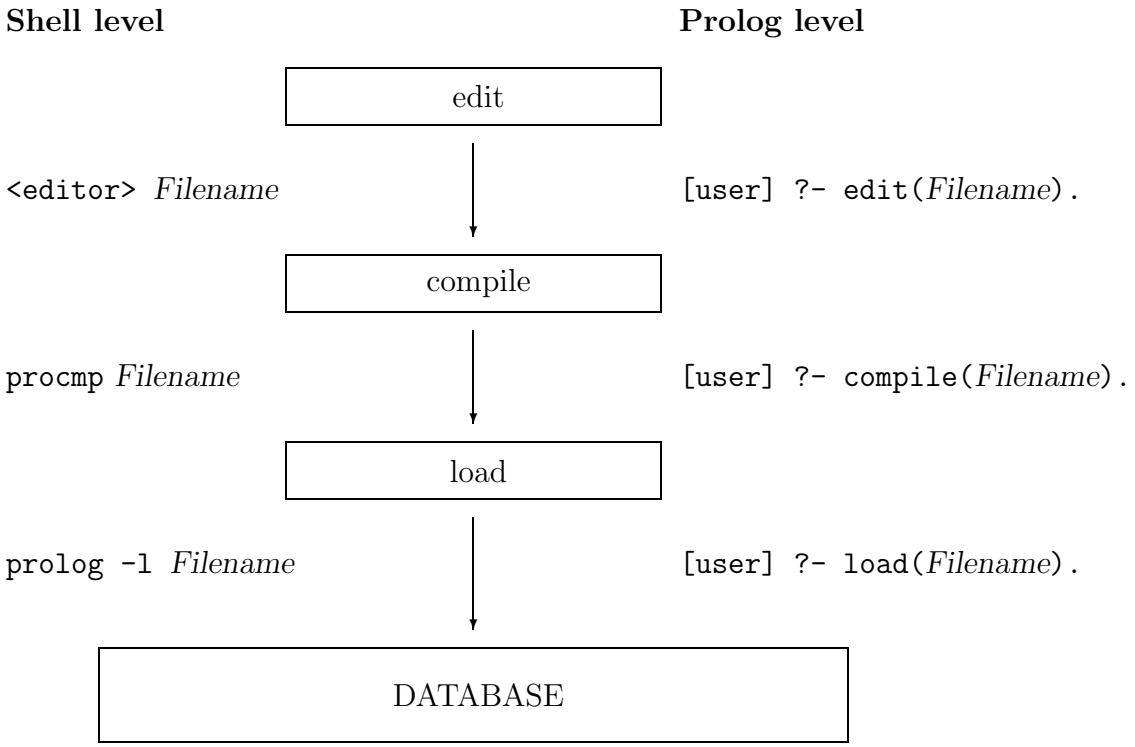
```
        system_function(Name, Arity),
        !,
        Function =.. [Name|Params],
        Result is Function.

evaluate_function(Name, _, Params, Result, Module) :-
        Function =.. [Name, Result| Params],
        once(Function) @ Module.

evaluate([], []).
evaluate([Arg|Args], [Param|Params]) :-
        Param := Arg,
        evaluate(Args, Params).

system_function((+), 2).
system_function((-), 2).
system_function((*), 2).
system_function((/), 2).

:- end_module expression.
```

The module `demo` uses the predicates of the module `expression` and the operator defined there.

```
:- module demo.
:- export [demo/0].

:- begin_module demo.
:- import expression.

:- op(100, xf, (!)).

demo :- N := 2! + 3!,
        writeq(result = N), nl.

'!'(R, N) :-
                N = 0
        ->
                R = 1
        ;
                N > 0, R := N * (N-1)!.

:- end_module demo.
```

7.5 Compiling and loading files

In addition to reading Prolog source texts, it is possible to compile the source texts into runtime-efficient code. The Prolog text which is contained in *Filename* is there stored in the internal form *Filename.cmp*. You may then load this into the database.



7.5.1 Compiling Prolog texts and modules

Compilation has several advantages over the interpretation of Prolog source texts:

- smaller runtime stacks
- protection against unauthorized copying of readable Prolog sources
- capability to restrict access permissions to a predicate
- reduced space requirement in the database

But there are also some disadvantages:

- certain restrictions apply to compiled predicates (see below); in particular, the compiled code cannot be modified while a program is running

- compiled code is not readable, and there are no `trace` or `debug` facilities.

You may mix compiled and interpreted predicates in the database as you wish.

Contents of a Prolog source file

To compile a Prolog source text, you should first have to write the text into a file with an editor. The file name must naturally comply with the naming conventions of the operating system.

You can compile any syntactically correct Prolog clause.

i All clauses of a predicate to be compiled must be stored in a single logical file unless it was declared with the multifile directive.

You can only perform compilation with reference to a module; any predicates which are compiled without specifying a module will automatically be assigned to the module `user`. You can also compile a file containing, for example, only module bodies. However, the corresponding module interfaces must then be known to the Prolog system, for example as a result of a previous compilation. Prolog then places the relevant complete module in the file which has been created.

A Prolog text file can also contain all the syntax elements (directives) required for the definition and declaration of modules.

Compiling a Prolog source file

There are two ways of compiling a Prolog source file *Filename*:

- Compilation at the operating system shell level by calling the compiler:

  $ procmp *Filename*

  Following a successful compilation the Prolog compiler automatically activates the optimizer (see also `proopt` in the *IF/Prolog Reference Manual* [1]).

  At the time of compilation the following module interfaces should be known to the Prolog compiler:

  - interfaces of the system modules from which you import predicates into the modules that are defined in *Filename*.
  - interfaces of other modules from which you import predicates to the modules that are defined in *Filename* and which are not defined in *Filename*.

- Compilation during a Prolog session with the predicate `compile/1`

  `[user] ?-` **compile(***Filename***).** ↩

Filename is the name of the Prolog source file. The result of the compilation is contained in *Filename.cmp*. If any suffix *.pro* is present in *Filename*, it will be removed. In contrast to compilation using `procmp`, the optimizer is not automatically activated. However, you can optimize the file manually with the `proopt` command.

At the time of compilation the following module interfaces must already be loaded:

- interfaces of the system modules from which you import predicates into the modules that are defined in *Filename*.
- interfaces of the modules from which you import predicates to the modules that are defined in *Filename* and which are not defined in *Filename*.

Possible errors

The system reports the following errors or exceptions:

- syntax errors in the Prolog source file
- read errors
- the file to be compiled does not exist
- the result file cannot be created (no access permission)
- insufficient space available

Restrictions on compiled predicates

If a predicate is to be modified or is still in the development or debugging phase, it is inadvisable to perform a compilation, for the following reasons:

- Compiled predicates cannot be modified.
  Following compilation, their definitions cannot be modified by (`assert...`, `consult/1`, `retract/1/2`, ...) predicates. If you attempt to make the predicate modifiable by calling the predicate `modify_mode/3`, the call will result in the exception `static_procedure`.
- You cannot inspect compiled predicates with the predicates `listing/0/1` or `clause/1/2` because the Prolog source text required for this is no longer available.
- Tracing or debugging of the compiled predicates is not possible.

What gets compiled?

Basically, the following applies: Only predicates defined in the specified text file are compiled. Undefined predicates (occurring for example as subgoals or arguments) are not compiled. In a file containing the definition

```
p(X):- g(X), asserta(remember(X)), fail.
```

`p/1` is compiled. The predicates `g/1`, `asserta/1`, `remember/1` are not compiled. This example also illustrates a good use of an `assert...` call within a compiled predicate.

Excluding predicates from compilation

By using the directive

```
:- dynamic PredicateList.
```

you may explicitly exclude predicates from the compilation. All predicates specified after `dynamic` are not compiled. Instead, they are merely inserted into the database, as with `reconsult/1`, when the file is loaded.
If your file contains the following Prolog source text:

```
:-dynamic [child/1].

child(anita).
child(sonja).
child(julia).

descendant(Name):-
        asserta(child(Name)),
        ...
```

you can compile the file; the predicate `descendant/1` is compiled. `child/1` is excluded from compilation. This makes it possible to modify `child/1` dynamically.

7.5.2 Loading compiled files

The result of a compilation is not linked into a main program before the program is started, as you might expect if you have worked with other languages such as C, Pascal, FORTRAN, etc. In Prolog, the compiled modules are read into the database after the Prolog system has been started. This operation corresponds approximately to the use of `reconsult/1` for non-compiled predicates, and is called **loading**.

All clauses and directives of a file that are not assigned to a module interface are assigned to the body of the built-in module `user`.

- The `initialization` directives are executed in the context of the relevant module.
- If the file contains a predicate that is already defined with the same functor and arity in the module `user`, it is replaced by the new definition as in the case of `reconsult/1`.
- A module that has already been loaded into the database is replaced in its entirety by the module definition in which a module with the same name is defined in the file.

After the file has been loaded, you can use the predicates in the same way as interpreted predicates. However, they can be neither modified nor analyzed with the debugger.

There are several ways of loading a compiled file into the database:

- by specifying an option when calling Prolog (see chapter 4.1.1, `-l` *Filename*)
- by using the predicate `load/1` during a Prolog session:

  `[user] ?-` **load(***Filename***).**↩

Filename is the name of the compiled Prolog file. Normally, *Filename* has the suffix *.cmp*. The system searches automatically in the following sequence:

- for a file called *Filename*
- if this is not found, for a file named *Filename.cmp*

and loads the file found.

Internal load operation

The file generated as a result of the compilation contains two types of code:

- Prolog source code for predicates excluded from compilation by means of `dynamic/1`
- compiled code for all other predicates.

Accordingly, loading a compiled file is internally divided up into:

- Loading all complete modules or module interfaces from the file. If the modules already exist in the database, they are replaced.
- Loading all clauses that are not assigned to a module from the file into the body of the module `user`. If predicates with the same name and arity are defined in the module `user`, these are replaced as in the case of `reconsult/1`. The file contents do not however, replace the entire module `user`.

- Execution of all `initialization` directives in the file in the context of the relevant modules. `initialization` directives that are not assigned to any module are executed in the context of the module `user`.

i Non-compiled predicates in the database will be overwritten by any compiled predicates with the same functor and arity when the latter are loaded. You cannot load a compiled file into the database with `consult/1` or `reconsult/1`.

Possible exceptions

Exceptions which may occur during loading include:

- The file *Filename* does not exist.
- Access to the file is not permitted.
- There is insufficient space in the database.
- The file was compiled using the compiler of an earlier version of IF/Prolog. The file does not therefore have the correct format for a compiled Prolog file.

An exception during loading causes the load operation to be aborted and an exception message to be generated.

7.5.3 Removing a module from the database

The predicate `unload/1` removes all information associated with the specified module from the database. Subsequently the module has neither an interface nor a body, but its name remains known.

`[user] ?-` **unload(***Module***).**↩

If the removed module was the current module, the built-in module `user` becomes the current module. If you remove the module `user`, however, Prolog immediately creates an empty module interface and an empty body again.

i You cannot remove built-in modules with `unload/1`.

Chapter 8

Contexts

A **context** is a logical environment for the execution of a goal, i.e. for the entire proof tree searched by a given goal, which may be a part of the entire proof tree of a Prolog application. Within a context, certain context-global **actions** may be executed. In setting up a context, information may be stored which will control or influence such context-global actions. The information is not destroyed until the context is exited upon completion of the goals execution. From the point of view of the entire application, the information is only local for the duration of the context.

There are no restrictions on the level of nesting of contexts. When context-global actions are executed, nested contexts are searched from inside to out for information which matches the context-global action.

There are various reasons why normal program execution is not possible, not relevant or not desirable. These include errors during goal execution, external events (such as signals) and possibly algorithmic requirements (global jumps, for example).

You can control program execution within a local program context. You can use highly differentiated exception and signal handling procedures with a high degree of reliability and independency within a local program context, thus excluding the reciprocal effects of different program sections which make use of global exception handling. IF/Prolog supports three classes of context-global action:

- Context jumps
- Exceptions
- Signals

The predicate `context(+`*Goal*`, +`*ContextInfo*`)` is used to create a context for the execution of *Goal*. The information for context-global actions specified in the list *ContextInfo* is stored in this context. How the information is structured and what the underlying semantics are is defined by the individual context-global actions.

Other predicates for context-global actions are:

Predicate	Purpose
catch(+*Goal*, +*CatchMask*, +*SubstituteGoal*)	Intercept a context jump
current_signal(?*Signal*, ?*Mode*)	Query signals and modes
exception(@*Error*, @*Info*)	Raise an exception
exception_handler(+*Goal*, +*Exception*, +*Handler*)	Define local exception handling
signal_control(+*Signal*, +*Command*)	Control signal processing
signal_handler(+*Goal*, +*Signal*, +*Handler*)	Define local signal handling
throw(@*Term*)	Initiate a context jump

The predicates for the context-global actions are all based on `context/2`. All predicates are described in the following sections.

8.1 Context jumps

In a context jump, (similar to a `longjmp(2)` in C, see [10]) a context is exited at any point in the proof tree and the entire proof tree is discarded at a stroke. When the context is exited, the actions defined at creation time can be executed.

You can explicitly initiate context jumps, or they may be executed automatically in the course of standard error handling in the system. If several contexts are nested, the innermost context matching the corresponding context jump is used. If no matching context is found, Prolog is exited.

The built-in predicate `break/0` (the Prolog input loop) defines a context which matches all the context jumps initiated in the course of standard error handling in the Prolog execution. Therefore, in the Prolog development system, an error will seldom be seen ever (apart from a very few exceptions) which leads to the Prolog system aborting.

You initiate a context jump with the predicate `throw(@`*Term*`)`. A call of `throw/1` is never evaluated with `true` or `fail`; instead, it always aborts the execution of current goals.

The Prolog system first searches, from inside to out, for an active context whose *ContextInfo* list contains an entry in the form `catch(`*CatchMask*`) =>` *SubstituteGoal* and in which *CatchMask* can be unified with a copy of *Term*. If such a context is found, the proof tree is discarded in this context, unification of *CatchMask* with a copy of *Term* is executed, the call of `context/2` relating to this context is replaced with *SubstituteGoal* and the execution is resumed. All the variable instantiations made in the course of execution of the goal within this context are undone, as is with `fail`. Only the instantiations created by the unification of *CatchMask* with a copy of *Term* are retained.

The *ContextInfo* list of the context may also contain multiple entries for each context-global action. When on the occurrence of a corresponding event, the first appropriate information entry is used. If no matching context can be found, Prolog aborts.

Example

Let us assume that the following predicate is defined:

```
test(H) :-
        H = test,
        throw(hello).
```

When it is called, the argument *H* is first unified with the atom `test`, next a context jump is initiated. If a context is defined and the predicate proved in this context, the instantiation of the argument of `test/1` is undone when the context jump is executed.

```
[user] ?- context(test(Z), [catch(Y) => true]).↩

Z       = _90
Y       = hello ↩

yes
```

The following predicate reads Prolog terms from a file and outputs the number of terms.

```
count_file(File) :-
        open(File, read, Stream),
        context(count_terms(Stream, 0),
                [ catch(end_of_file(Count)) => count_write(Count)]),
        close(Stream).

count_terms(Stream, Count) :-
        read(Stream, Term),
        check_term(Term, Count),
        NewCount is Count + 1,
        count_terms(Stream, NewCount).

check_term(end_of_file, Count) :-
        throw(end_of_file(Count)).
check_term(_, _).

count_write(Count) :-
        write(Count),
        write(' terms counted.\n').
```

There is a special predicate for context jumps which creates a context which only handles context jumps as context-global actions. The predicate `catch(+`*Goal*`, +`*CatchMask*`, +`*SubstituteGoal*`)` creates a context for the execution *Goal*. If, in the course of execution *Goal*, the goal `throw(@`*Term*`)` is executed, where *Term* may be unified with *CatchMask*,

goal execution is terminated and the call of `catch/3` is replaced by *SubstituteGoal*. The call `catch(`*Goal, CatchMask, SubstituteGoal*`)` is therefore identical to `context(`*Goal*`, [ catch(`*CatchMask*`) => `*SubstituteGoal*` ])`.

You can use the predicate `catch/3` or the corresponding `context/2` call to resume program execution at a defined point after errors. Use is made of the fact that the default exception handling procedure of the Prolog system consists actually of executing a context jump.

Example

If, for example, a test is to be carried out to determine whether an argument in a predicate is an arithmetic expression and depending on the result of this test a different action is to be executed, this may be implemented with `catch/3` as follows:

```
test(Arg) :-
        catch(Result is Arg, _, fail),
        first_action(Result).

test(Arg) :-
        other_action(Arg).
```

i When *SubstituteGoal* is proved after execution of a context jump, the context for the original goal execution is already destroyed since *SubstituteGoal* replaces the corresponding `context/2` call. This means that program cycles cannot occur when context jumps are intercepted by `catch/3` or `context/2`, so this is a very save method of error handling.

8.2 Exceptions

Exceptions are raised by the Prolog system if errors have occurred. However, you can also raise exceptions explicitly with the predicate `exception(@`*Error*`, @`*Info*`)`. They are handled exactly as if they were exceptions raised by the Prolog system.

If one of the active contexts provides information on how to handle the exception, exception handling is executed without destroying the current proof tree in the context. Exception handling replaces the goal which initiated the exception. At the end of the exception handling, the goal execution is continued normally, unless a context jump is executed during the exception handling.

If no active context is found which can handle the exception, standard exception handling is executed by the Prolog system. Standard exception handling performs a context jump.

Each exception is described by the type of the error and possibly additional information.

Error types

The error type *Error* is an atom which describes an error more precisely. The following error types are used by the Prolog system:

Error type	Meaning
domain_error	The argument does not lie in the required value range.
evaluation_error	An arithmetic expression causes an error during evaluation.
existence_error	An object to which an operation is to be applied is not present.
instantiation_error	An argument or subargument is a variable.
permission_error	The execution of an operation for an object is not permitted.
representation_error	The implementation-dependent restriction has been exceeded.
resource_error	The resources are inadequate for further processing.
syntax_error	A character string which is to be read in as a term, does not conform to the Prolog syntax.
system_error	An operating system error has occurred.
type_error	An argument or subargument does not conform to the expected type.

With most of the error types, the Prolog system also supplies information which specify the error cause more precisely and the culprit error term which caused the error.

Additional information

Info is a list whose elements have the form *InfoName* = *Value* and which supplies additional error-related information. *InfoName* is an atom which determines the type of the additional information. The Prolog system can supply the following additional information:

Additional information	Meaning
argno	Argument number relating to the current Prolog goal which caused the error.
column	Column where a syntax error has occurred.
culprit	Term which caused the error.
error	Atom indicating the type of a system error, e.g. `fork_failed`
goal	Goal which initiated the error.
interrupted	If a predicate implemented in C was interrupted by a signal, this information is contained here.
line	Line where a syntax error has occurred.
signal	Name of the signal.
syntax_error	Type of a syntax error.
valid_type	Expected type of a term.

The order of the additional information is not defined. You can also expand the *Info* list with additional information of your own. However, you should only use the predefined designators in a consistent manner.

Example

If you attempt to open a non-existent file with the name *hello* for reading

```
[user] ?- open(hello, read, Stream).↩
```

the Prolog system raises the following exception:

```
*** E X C E P T I O N: existence_error(source_sink,hello)
>>> error       = existence
>>> goal        = system : open(hello,read,_225)

no
```

When an exception is raised, a search is carried out, from inside to out, for an active context whose *ContextInfo* list contains an entry in the form `error(`*Error*`, `*Info*`) => `*Handler* where *Error* and *Info* can be unified with the corresponding information of the initiated exception. If such a context is found, the goal which initiated the exception is replaced by *Handler*. Goal execution is then resumed as normal.

If no matching context is found, the goal initiating the exception is replaced with the goal `throw( error(`*Error, Info*`) )` and executed; in other words a context jump is performed. You can intercept this with `catch/3` or a corresponding `context/2` call.

i The Prolog input loop (`break/0`) defines a context which catches all exceptions and generates appropriate error outputs from the information supplied.

Example

The following Prolog text implements a simple read-evaluate-print loop similar to `break/0`.

```
mybreak :-
        repeat,
        catch(
                (read(Goal), eval(Goal)),
                error(E, A),
                exception_message(E, A)
        ),
```

```
        fail.

eval(end_of_file) :-
        halt.

eval(Goal) :-
        call(Goal).

exception_message(Error, Message) :-
        write_formatted('*** E X C E P T I O N: %s(%s)\n', Error),
        write_formatted('*** %q in %q\n', [ Message ]).
```

There is a special predicate which creates a context which handles only exceptions. The predicate `exception_handler(+`*Goal*`, +`*Exception*`, +`*Handler*`)` creates a context for *Goal*. If, while *Goal* is being proved an exception is raised whose description can be unified with *Exception* as the term `error(`*Error*`, `*Info*`)`, the goal which raised the exception is replaced by *Handler* and execution is resumed. The call `exception_handler(`*Goal*`, error(`*Error, Info*`),` *Handler*`)` is therefore identical to `context(`*Goal*`, [error(`*Error, Info*`) => `*Handler*`])`.

The predicate `context/2` has the advantage that different types of events can be handled in one context.

i Do not attempt to handle **all** exceptions in a single context because then there is a danger of endless loops. You should always handle possible exceptions specifically.

Example

The following Prolog text implements a simple autoload mechanism for predicates which are defined in a collection of facts.

```
:- private pred1/3, pred2/2, pred3/1.

auto_load(pred1/3, 'hello.pro').
auto_load(pred2/2, 'preds.pro').
auto_load(pred3/1, 'hello.pro').

mybreak :-
        context(break,
        [ error(existence_error(procedure, Pred), Info)
          => try_autoload(Pred, Info)]).

try_autoload(Pred, Info) :-
        auto_load(Pred, File),
        member(goal=Goal, Info),
```

```
        consult(File),
        predicate_type(Pred, Type),
        Type \= undefined,
        !,
        Goal.

try_autoload(Pred, Info) :-
        throw(error(existence_error(procedure, Pred),Info)).
```

The handling of exceptions takes place within the context which defines this handling. Consequently there is a danger of cycles if exceptions occur during handling itself and are then handled in the same context. The following rules may help you to avoid such situations:

1. You should specify exceptions to be handled as precisely as possible so that the context handles only the exceptions it is intended to handle. There will be very few cases where it is appropriate to specify that all exceptions are to be handled.

2. When an exception is handled, you should ensure that the cause of the exception is eliminated so that the exception does not immediately re-occur. In the previous example, this is achieved by checking before calling *Goal* again, whether the predicate is defined after the load operation.

3. You should pay special attention to planning what is to happen when during handling, it is established that the handling cannot be successfully performed. In particular, the original exception should not be generated again. In the previous example this is covered by the second clause of `try_autoload/2`. No exception is generated; instead, a context jump is executed as in standard exception handling.

8.3 Signals

Signals are asynchronous events arriving from the outside world. One of the ways in which signals can be dealt with is active handling in the application. If there is provision for signal handling by the application and if information is provided for in one of the active contexts on how to handle the signal, handling is performed without destroying the current proof tree.

If signal handling is successful, the goal interrupted by the signal is resumed as if the signal had not arrived. Otherwise, or if no context is found which can handle the signal, the exception `error(system_error, [error=interrupt, signal=`*Signal*`])` is generated and a context jump is executed.

In IF/Prolog, the way in which the system reacts to the signals it receives can be controlled selectively. The following signals can be handled (in some operating systems, some signals are not supported):

Prolog signal	C signal name (see [10])
abort	SIGABRT
alarm	SIGALRM
interrupt	SIGINT
pipe	SIGPIPE
quit	SIGQUIT
termination	SIGTERM
user_1	SIGUSR1
user_2	SIGUSR2

For each of these signals it is possible to define the general handling mode. The following handling modes are provided:

- Default handling
  is the standard handling provided by the operating system for the signal (program abort in most cases).
- Ignore
  The operating system is instructed to ignore arriving signals. The Prolog system does not receive these signals at all.
- Passive handling
  by the Prolog system; this means that the arrival of the signal is registered but it does not affect program execution.
- Active handling
  by the application or the Prolog system. If the application provides signal handling, the routines are executed; otherwise the Prolog system takes control and initiates a context jump (`throw/1`).

  As with exceptions, active handling of a signal by the application is possible only within a local context.

With the predicate `current_signal/2` you can query the signal modes which were active on the first call, even if they have changed in the meantime (**logic update view**).

Example

```
[user] ?- current_signal(S, M), write(S : M), nl, fail.↩


user_2 : default
user_1 : default
pipe : default
termination : default
```

```
quit : default
alarm : default
abort : default
interrupt : on

no
```

8.3.1 Global control of signal handling

You can selectively activate, disable or delay signal handling for individual signals. The predicate `signal_control/2` allows you to globally control signal handling; you may specify the following commands for a signal:

Mode	**Meaning**
clear	Handling of signals which have been received but not yet processed is suppressed. The number of signals still to be processed is set to 0.
default	When the signal is received, the actions provided by the operating system are performed (normally program abort; if `abort` and `quit` occur, a memory dump can also written to disk).
ignore	Signal handling by the operating system is disabled for the signal.
on	Active signal handling is enabled for a signal; i.e. when a signal arrives, either user handling is activated or an exception is initiated.
off	When the signal is received, it is handled passively by the Prolog system, i.e. the number of incoming signals is counted, but no action is taken. If signal handling is reactivated with `on`, signal handling is also performed for registered signals.
received	Queries whether the signal has been received at least once, i.e. whether the counter for this signal is greater than 0. In this case, the counter for the arrival of the signal is decremented by 1.
received(*Name*)	Queries whether the signal has been received at least once. The counter for the arrival of the signal is decremented by 1 here if it was not 0. *Name* is unified with the name of the signal received.

Immediately before a handler predicate for the active signal handling of *Signal* is called, the command `off` is set for the signal. The signal handling routine is itself responsible for the renewed activation of the signal (command `on`). This is done to prevent the signal handler routine from being interrupted by the re-occurrence of the signal.

After Prolog is called, default signal handling (`default`) of the operating system is activated for all signals. If the Prolog input loop (`break/0`) is started, handling of the `interrupt` signal is set to `on`.

8.3.2 Passive signal handling

If you activate passive signal handling for a signal with `signal_control/2` and this signal arrives, then an internal counter for the arrival of this signal is incremented by 1 and processing continues. You may query or clear this counter with `signal_control/2`.

Example

The following short example implements a reaction test:

```
reaction :-
        for(1, _, 10000),
        fail.

reaction :-
        signal_control(interrupt, off),
        signal_control(interrupt, clear),
        write('Press interrupt key! '),
        flush_output,
        for(1, Cnt, 30000),
        signal_control(interrupt, received),
        nl,
        write('Result: '),
        write(Cnt), nl,
        !.

reaction :-
        nl,
        write('too slow'), nl.
```

The first of the two clauses is used as a header. At the end of this loop, `Press interrupt key!` is displayed and you now have to hit the interrupt key (e.g. DEL) as quickly as possible. The smaller the result value displayed, the faster your reaction.

```
[user] ?- reaction.↩
Press interrupt key!  DEL
Result:  1269


yes
```

i In certain cases, the arrival of signals can lead to irreparable damage. There are a number of **system calls** which can be interrupted by the arrival of signals. This may occur in particular with operating system services which lead to a program wait state, for example read operations which may have to wait for user input (from the keyboard), e.g. the predicate `read/1/2`.
In such cases, passive signal handling also leads to changes in program execution.

8.3.3 Active signal handling

If a signal arrives for which you have activated active signal handling with `signal_control/2`, Prolog searches, from the inside out, for an active context whose *ContextInfo* list contains an entry in the form `signal(`*Signal*`) =>` *Handler*, where *Signal* is unified with the name of the received signal. If Prolog finds such a context, it proves *Handler* without destroying the current proof tree. If *Handler* is successful (`true`), the interrupted goal is resumed as if the *Signal* had never arrived. Otherwise, or if Prolog finds no matching context, it generates the exception `error(system_error, [error=interrupt, signal=`*Signal*`, goal=`*Subgoal*`])`, i.e. initiates a context jump, where *Subgoal* is the goal which would have been proved next if the signal had not arrived.

There is a special predicate which creates a context which handles only signals. The predicate `signal_handler(+`*Goal*`, ?`*Signal*`, +`*Handler*`)` creates a context for executing *Goal*. If, while *Goal* is being proved, a signal arrives for which active handling is defined and whose name can be unified with *Signal*, the goal *Handler* is proved at the interrupt point. The call `signal_handler(`*Goal, Signal, Handler*`)` is therefore identical to `context(`*Goal*`, [signal(`*Signal*`) =>` *Handler*`])`.

The predicate `context/2` has the advantage that different types of events can be handled in one context.

Example

The following predicate defines a daemon which becomes active when the signal `user_1` arrives and waits for further signals at the end of the activity. The daemon terminates when the signal `user_2` arrives.

```
daemon(WorkGoal) :-
        signal_control(user_1, on),
        signal_control(user_2, on),
        context(sleep, [
                catch(end_of_work) => true,
                signal(user_1) => work(WorkGoal),
                signal(user_2) => throw(end_of_work)]).
```

```
sleep :-
        repeat,
        fail.

work(WorkGoal) :-
        call(WorkGoal),
```

Chapter 9

Debugging

This chapter describes the debugging facilities, which consist of a tracer and an interactive debugger. These tools assist the programmer when debugging and correcting Prolog programs. The programmer can:

- monitor the sequence of execution
- halt execution
- intervene in the processing
- query information

This has the advantage of allowing clause selection to be monitored in addition to subgoal calling and exiting. The debugging tools are based on the extended box model which is described in chapter 3.

9.1 Debugger settings

Before starting debugging activities, you can use built-in predicates to set certain controls which are to apply to the tracer or the interactive debugger:

- `debug_mode/3` defines the debug mode of a predicate
- `debug_config/3` configures the debugger
- `trace_config/3` configures the trace.

The predicates are described at the end of this chapter.

9.1.1 Activating/deactivating box ports

The debugging tools enable you to follow the progress of goal execution with the box model. Each of the thirteen ports of the extended box model serve as a possible control point. Before starting debugging, you can select which box ports are to be activated by default for your debugging session. The port settings apply to both the tracer and the interactive debugger. To select the ports, you use the built-in predicates `debug_config/3` or `trace_config/3`.

The following table shows the default settings (in boldface) for the debugger:

Parameter	Value	Meaning
call	**on**	Debugger: Breakpoint at the call port
	off	Trace: Activate/deactivate listing for the call port
error	**on**	Debugger: Breakpoint at the error port
	off	Trace: Activate/deactivate listing for the error port
exit	**on**	Debugger: Breakpoint at the exit port
	off	Trace: Activate/deactivate listing for the exit port
fail	**on**	Debugger: Breakpoint at the fail port
	off	Trace: Activate/deactivate listing for the fail port
redo	**on**	Debugger: Breakpoint at the redo port
	off	Trace: Activate/deactivate listing for the redo port
wakeup	**on**	Debugger: Breakpoint at the wakeup port
	off	Trace: Activate/deactivate listing for the wakeup port
suspend	**on**	Debugger: Breakpoint at the suspend port
	off	Trace: Activate/deactivate listing for the suspend port
trymatch	on	Debugger: Breakpoint at the trymatch port
	off	Trace: Activate/deactivate listing for the trymatch port
failmatch	on	Debugger: Breakpoint at the failmatch port
	off	Trace: Activate/deactivate listing for the failmatch port
enterbody	on	Debugger: Breakpoint at the enterbody port default: **off**
	off	Trace: Activate/deactivate listing for the enterbody port default: **on**
exitbody	on	Debugger: Breakpoint at the exitbody port default: **off**
	off	Trace: Activate/deactivate listing for the exitbody port default: **on**
redobody	on	Debugger: Breakpoint at the redobody port
	off	Trace: Activate/deactivate listing for the redobody port
failbody	on	Debugger: Breakpoint at the failbody port
	off	Trace: Activate/deactivate listing for the failbody port

Example

```
[user] ?- debug_config(exitbody, _, on).↩
```

activates the `exitbody` port in addition to the external ports. All other internal ports remain deactivated.

9.1.2 Debug mode of a predicate

The debug mode of a predicate allows you to define whether a predicate can be debugged. The debug mode of a predicate applies during both tracing and interactive debugging.

- Predicates with debug mode turned on can be monitored, i.e. the sequence of execution within the predicate box (clause unifications and subgoals) can be debugged.
- Predicates with debug mode turned off cannot be monitored, i.e. execution sequences within the predicate box cannot be debugged.

You should also note the following:

- C predicates and compiled predicates of a module cannot be monitored.
- Non-compiled user-defined predicates can be monitored by default. With this type of predicate, you can use `debug_mode/3` to switch off debug mode, to switch it back on again, and to query it:

```
[user] ?- debug_mode(Predicate, Old_Mode, off).↩
[user] ?- debug_mode(Predicate, Old_Mode, on).↩
[user] ?- debug_mode(Predicate, Mode, Mode).↩
```

i You should note that the Prolog system regards predicates as dynamic or modifiable only if either you have called Prolog with the `-debug` parameter or have declared the predicates as dynamic with the `dynamic` directive.

9.1.3 Further parameters for the debugger

The predicates `debug_config/3` and `trace_config/3` allow you not only to define breakpoints for the debugger and tracer, but also to control the output:

Parameter	Value	Meaning
display_depth	7	Maximum structure depth in which terms are output. If you specify the value 0 here, terms are displayed down to the lowest level.
history_length	3	Number of goal and clause boxes which are to be stored. This parameter applies only to the debugger.
trace_length	100	Number of trace outputs which are to be stored. This parameter applies only to the trace.
display		Option for setting the user interface. This parameter applies only to the debugger.
	automatic	The user interface is set automatically by the debugger according to the system environment.
	alpha	The user interface is set for alphanumeric screens. The input/output device must be a terminal and the device driver must recognize the terminal control functions. If this is not the case, the debugger switches over automatically to a stream user interface.
	stream	The debugger switches on the command-line stream user interface. This user interface requires no terminal functions and is consequently also suitable for redirection of input/output.
	motif	The debugger attempts to start the OSF/Motif user interface on the preset display. If the attempt is unsuccessful, the debugger switches back automatically to the alphanumeric user interface.
	motif(*Display*)	The debugger attempts to start the OSF/Motif user interface on the specified *Display*. If the attempt is unsuccessful, the debugger switches back automatically to the alphanumeric user interface.

9.2 The Tracer

The tracer can provide you with more detailed information about the search for a goal, but does not allow you to intervene in the execution of the search.

9.2.1 Predicates for the trace

The following table summarizes all predicates which are relevant to tracing.

Predicate	Purpose
debug_mode(@*Predicate*, ?*Before*, +*After*)	Query and set the debug mode of a predicate
trace(+*Goal*)	Activate tracing for a goal
trace_begin	User-defined tracer initialization
trace_config(+*Parameter*, ?*Old*, +*New*)	User-defined tracer initialization
trace_end	User-defined termination of the tracer
trace_output(@*CallNumber*, +*Port*, @*Goal*, @*VariableNames*)	User-defined output predicate for the tracer

The predicates are described in full in the *IF/Prolog Reference Manual* [1]. However, the main features of the tracer are covered in the sections below.

9.2.2 Start trace

You can turn on the trace alone or activate it under the debugger:

- In the top-level dialog you turn on the trace with the command `trace(`*Goal*`)`. Before the trace logs the proof of *Goal*, the predicate `trace_begin/0` is called. After proof of the goal, the predicate `trace_end/0` is executed. You can define both predicates yourself.
- The debugger and the trace can be on simultaneously. You call a goal whose processing you wish to trace by means of the debugger command `trace(on)`. In this case, the two predicates `trace_begin/0` and `trace_end/0` are not executed.

9.2.3 Trace operation

When the trace is on, a step-by-step execution listing is generated and output. An execution step is represented by the appropriate box port. All ports permitted by the debug mode of the predicate can be monitored.

The trace listing is written to the current error output stream. You can redefine this stream if necessary. All activated ports (default: `call, fail, exit` and `redo`) are traced.

Output format	An output listing contains the following columns for each execution step: •box port which represents the execution step; •subgoal with which the execution step is associated.
Standard format	The column entries are listed side by side, with each execution step on a separate line.
User control facilities	You can use `trace_output/4` to structure a listing of your own.

Example

In order to demonstrate the debugging facilities we shall use the following predicates in most of the examples:

```
father(max,gregor).
father(max,sabine).
father(max,rudi).
father(gregor,hans).
father(gregor,pepsi).
father(hans,olaf).

mother(sabine,eva).
mother(sabine,anita).
mother(sabine,ludwig).
mother(eva,suzi).
mother(pepsi,miki).
mother(miki,ulf).

ancestor(X,Y) :- father(X,Y).
ancestor(X,Y) :- mother(X,Y).
ancestor(X,Y) :- father(X,Z), ancestor(Z,Y).
ancestor(X,Y) :- mother(X,Z), ancestor(Z,Y).
```

```
[user] ?- trace(ancestor(X,hans)).↩


1.1      call           call(ancestor(X,hans)) @ user
2.1      call           ancestor(X,hans)
3.0      enterbody      ancestor(X,hans)
3.1      call           father(X,hans)
4.0      enterbody      father(gregor,hans)
4.1      call           true
4.1      exit           true
4.0      exitbody       father(gregor,hans)
3.1      exit           father(gregor,hans)
3.0      exitbody       ancestor(gregor,hans)
2.1      exit           ancestor(gregor,hans)
1.1      exit           call(ancestor(gregor,hans)) @ user

X       = gregor ;


1.1      redo           call(ancestor(gregor,hans)) @ user
2.1      redo           ancestor(gregor,hans)
3.1      redo           father(gregor,hans)
```

```
3.1      fail           father(X,hans)
3.0      enterbody      ancestor(X,hans)
3.1      call           mother(X,hans)
3.1      fail           mother(X,hans)
3.0      enterbody      ancestor(X,hans)
3.1      call           father(X,Z)
4.0      enterbody      father(max,gregor)
4.1      call           true
4.1      exit           true
4.0      exitbody       father(max,gregor)
3.1      exit           father(max,gregor)
3.2      call           ancestor(gregor,hans)
4.0      enterbody      ancestor(gregor,hans)
4.1      call           father(gregor,hans)
5.0      enterbody      father(gregor,hans)
5.1      call           true
5.1      exit           true
5.0      exitbody       father(gregor,hans)
4.1      exit           father(gregor,hans)
4.0      exitbody       ancestor(gregor,hans)
3.2      exit           ancestor(gregor,hans)
3.0      exitbody       ancestor(max,hans)
2.1      exit           ancestor(max,hans)
1.1      exit           call(ancestor(max,hans)) @ user

X       = max

yes
```

9.2.4 Stop trace

If you call `trace/1` when the debugger is not active, tracing will terminate of its own accord after the specified goal has been proved. The user-defined predicate `trace_end/0` is called and then the trace is turned off. When the debugger is active, the debugger command `trace(off)` stops the trace.

9.3 Interactive debugger

Simply tracing the execution of your programs does not usually give you enough information for pinpointing specific errors or for thorough debugging. Prolog therefore provides you with an interactive debugger which allows you to do the following:

- Execution monitoring

You can select various types of breakpoints which enable you to monitor the execution of a goal in small or large steps.

- Execution control
  You can return to previous execution points, skip execution steps, or select normal execution.

- Execution intervention
  You can intervene to perform other actions during the processing of a predicate.

- Querying information

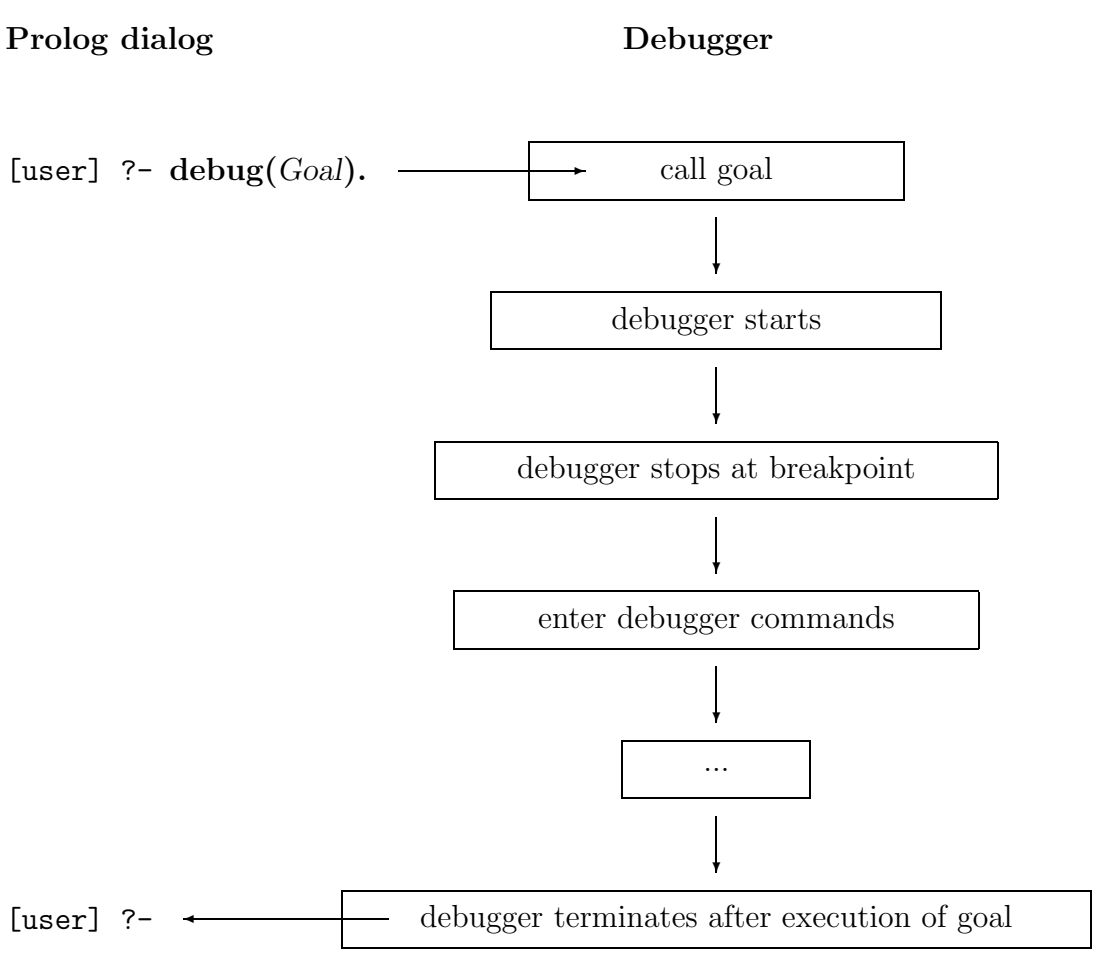
Before you debug a predicate, you should check whether you are allowed to do so. If you call the Prolog system without parameters, all predicates are compiled by the incremental compiler and thus held as static predicates in the database. You are not permitted to debug static predicates.

You define all predicates as dynamic if you call the Prolog system with the `-debug` parameter. You can declare individual predicates as dynamic with the `dynamic` directive.

9.3.1 Debugger operation

Depending on its settings, the debugger executes a proof for a goal from one port to the next, checking at each step whether a breakpoint is set. When it encounters a breakpoint or an error, the debugger stops, outputs the current status in the appropriate input/output areas and is ready to execute commands. You may then enter debugger commands in order to request information, change debugger settings or continue execution.

Schematically, a debugger session proceeds as follows:



You can only call the debugger for a particular goal:

```
[user] ?- debug(ancestor(X, pepsi)).↩
```

After the call of `debug/1`, the debugger starts and first calls the predicate `debug_begin/0`, if defined by the user.

Debugger settings

The debugger provides default settings. You can change settings by means of predicates and debugger commands in order to define your own debugging strategy:

Predicate or command	Meaning
debug_config/3	Configure debugger
trace_config/3	Configure trace
port	Activate/deactivate ports for debugger
trace_port	Activate/deactivate ports for trace
set_debug_depth	Define structure depth for debugger
set_trace_depth	Define structure depth for trace
set_trace_length	Set length of trace field
set_history	Set number of clause and goal boxes to be stored
stop	Set explicit breakpoints
stop_interactive	Set explicit interactive breakpoints

The settings remain in effect until you explicitly change them.

By defining the predicates `debug_begin/0` and `debug_end/0`, you can specify start and end processing operations which are to be performed automatically whenever the debugger starts and exits.

You can temporarily exit the debugger:

- by using the debugger command `B` or `break` to start a new Prolog loop. You can then work in the Prolog dialog until you enter `end_of_file`. The Prolog system then returns automatically to the debugger.
- by using the command `call`, or its short form `ca`, to call a goal outside the control of the debugger.
- by using the command `system`, or its short form `S`, to start an operating system subshell outside the Prolog system.

When the debugger has proved the test goal, it terminates itself automatically and you are returned to the Prolog top-level dialog. The top-level dialog prompt appears on the screen:

```
[user] ?-
```

9.3.2 Breakpoints

Breakpoints are always associated with the ports of the box model. The debug mode of a predicate determines which ports are permitted as breakpoints. Where a predicate is monitorable, all thirteen box ports are possible breakpoints. Where a predicate is not monitorable, you are not allowed to examine the box for that predicate. Consequently, only the `call, error, exit, fail, redo, suspend` and `wakeup` ports may be used as breakpoints. If you try to specify other ports, the debugger ignores them.

Provided a port is active by the debug mode set for the predicate, the debugger stops when

- the port is activated
- an explicit breakpoint is set at this port and its call is successful at this port
- the port is the destination of a branch command (see page 173).

You use breakpoints to interrupt the execution of a goal and to perform certain actions; three types of breakpoint may be differentiated:

- implicit breakpoints
- explicit breakpoints
- temporary breakpoints

Breakpoints which are skipped by commands are also referred to as deactivated breakpoints in the following.

Implicit breakpoints

Implicit breakpoints are points at which the debugger switches to interactive mode. You define implicit breakpoints before calling the debugger with the predicate `debug_config/3` or during the debugger session with the command `port/2` or the menu `Ports`.

Debugger commands such as `nonstop` or `fast_skip` skip the implicit breakpoints.

Explicit breakpoints

You set explicit breakpoints yourself during a debugger session. In doing so you can set simple explicit breakpoints or make stopping in the debugger dependent on conditions.

From the static viewpoint, a Prolog program consists of predicates, and these in turn of clauses. A clause is a fact or a rule, comprising of a clause head and subgoals. You can define explicit breakpoints for:

- the entire predicate
- the individual clauses
- the subgoals of the clauses
- a variable in a clause

You can have the instantiation of a variable output at an explicit breakpoint. If this monitored variable is part of a structure which also contains variables, you can also extend monitoring to these variables.

Explicit breakpoints are further differentiated into interactive and non-interactive breakpoints, depending on whether or not you want the debugger to switch to interactive mode at the breakpoint. You use the commands `stop/4` or `stop_interactive/1/2/4` to set an explicit breakpoint. The following table lists all the commands with which you can process explicit breakpoints.

Command	**Short form**	**Meaning**
activate_stop	as	Activate explicit breakpoint
deactivate_stop	ds	Deactivate explicit breakpoint
remove_stop	rs	Delete explicit breakpoint
stop	s	Set explicit breakpoint
stop_interactive	si	Set explicit interactive breakpoint
view_stops	vs	Display breakpoints

The commands are described in the reference section in the *IF/Prolog Reference Manual* [1].

When you set a breakpoint, you can use conditions to define whether or not the debugger is to stop at an explicit breakpoint:

- `frequency` gives the number of passes through a breakpoint. The number is incremented by 1 whenever the breakpoint is reached. By formulating a `frequency` condition in the commands `stop/4` and `stop_interactive/1/2/4` you can specify that a port is to become a breakpoint only after a certain number of passes.

  ```
  stop_interactive(p/1,call,frequency>5,[])
  ```

- `depth` gives the current tree depth in the search tree. By formulating a `depth` condition on setting an explicit breakpoint you can specify that a port is to become a breakpoint only at a certain tree depth.

  ```
  stop_interactive(p/1,call,depth>3,[])
  ```

- `recursion` gives the current recursion depth during execution of a recursive predicate. The recursion depth is incremented by 1 whenever the predicate is called or is called again recursively (`call, redo`). The recursion depth is decremented by 1 whenever a recursive call returns (`fail, exit`). By formulating a `recursion` condition on setting an explicit breakpoint you can specify that a port is to become a breakpoint only at a certain recursion depth.

  ```
  stop_interactive(p/1,call,recursion==3,[])
  ```

- `match(Head)` checks at the time of setting the breakpoint whether the *Head* of the specified clause can be unified. If this check fails, the breakpoint is not set. If unification

is possible, formulating a `match(Head)` condition will cause the debugger to stop at the breakpoint when *Head* is unifiable with the current subgoal.

```
stop_interactive(p/1,call,match(p(3)),[])
```

Temporary breakpoints

You cannot explicitly create temporary breakpoints. You create a temporary breakpoint indirectly, for example by means of a branch command or if goal execution is exited through an `exit, fail` or `error` port. Temporary breakpoints are always interactive breakpoints.

For all these ports the debugger creates a set of temporary breakpoints; these are created jointly and are also deleted jointly. If you issue a command which creates a temporary breakpoint before one of the most recently created breakpoints has been encountered, the old temporary breakpoints are deleted before the new ones are created.

9.3.3 Controlling debugger execution

Starting from the current breakpoint, you can control subsequent execution of the debugger. The first breakpoint for a goal is the `call` port.

Command	**Short form**	**Meaning**
continue	c	Execute to next explicit breakpoint
fast_skip	fs	Execute to exit of current subgoal
nonstop	no	Execute to end
skip	s	Execute to exit of current subgoal
step	↩	Execute to next breakpoint

As soon as the debugger reaches a port of the box model it checks whether breakpoints are set at this port and if so, whether they are activated. If no breakpoint is present, the debugger continues the execution in accordance with the current execution command.

If breakpoints are present at the current port, as its first step the debugger executes all the commands which you have associated with this breakpoint. If one of these commands causes the debugger to continue the execution, it will be immediately executed. The debugger thus leaves the breakpoint and any other commands present at this breakpoint will have no affect.

After execution of all commands, the debugger switches to interactive mode if at least one of the breakpoints was interactive. However, the debugger switches only once to interactive mode, regardless of the number of interactive breakpoints.

9.3.4 Branching in the proof tree

Return branches allow you to branch back in the proof tree within the debugger. You can thus place the debugger in a state which has already been reached, without the necessity for a breakpoint at the port to which the debugger is to branch.

Cross branches are box-related manipulative branches which place the execution in a state which it would not generally attain in the course of normal program execution. You can use cross branches to simulate success, failure or the occurrence of an error during the proof of a subgoal or a clause.

Command	**Short form**	**Meaning**
back_clause	bc	Branch back to clause head
back_parent	bp	Branch back to `call` port of calling goal
back_subgoal	bs	Branch back to `call` port of subgoal
error	err	Generate exception
exit	x	Force current goal to succeed
fail	f	Force current goal to fail

The debugger stops at the branch destination regardless of whether the port is activated. If a branch destination does not exist, an error message is issued. Branch commands allow you to:

- check execution steps retrospectively by branching back to a preceding execution point.
- determine the behavior of subgoals for test purposes using cross branches to specify how a box is to be exited.

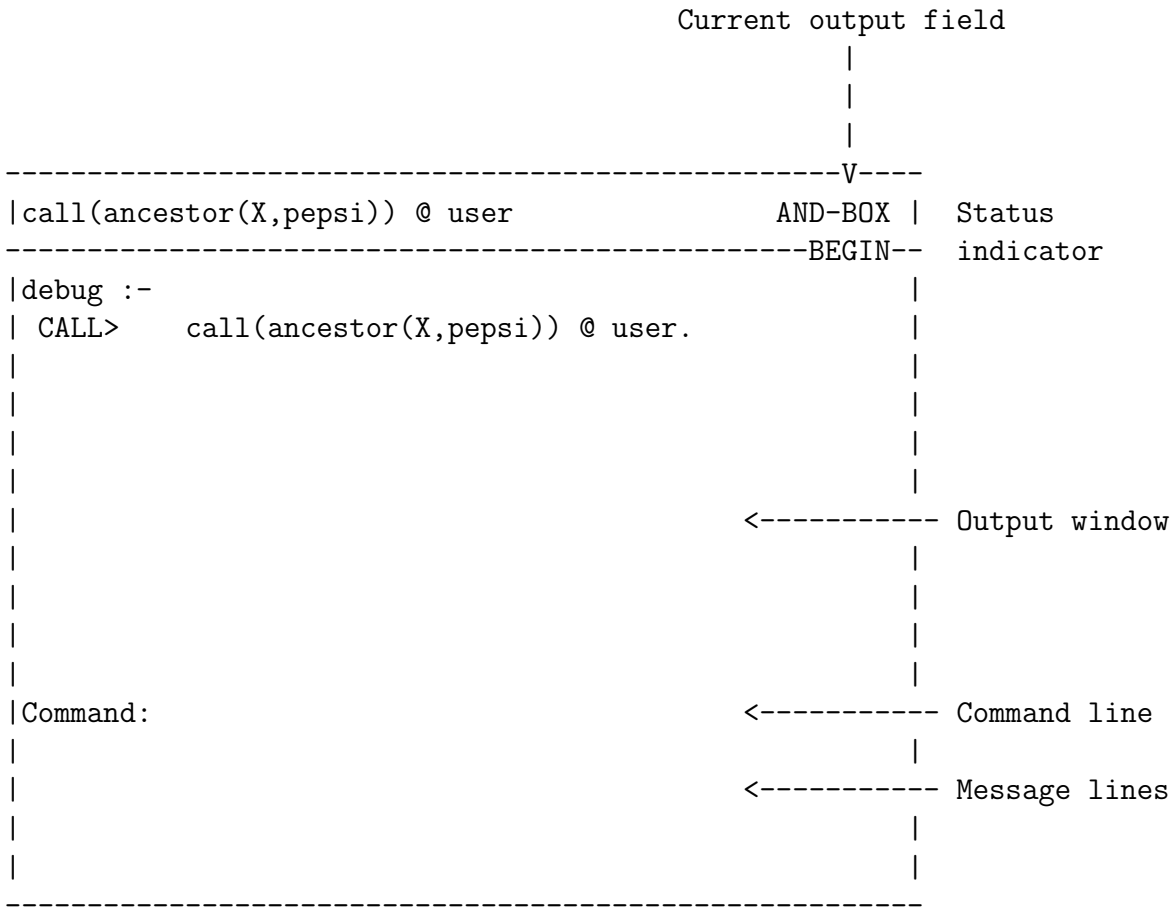
The debugger command `view_ancestors` is of interest in connection with branch commands as it allows you to display the dynamic ancestors.

9.4 Debugging with the alphanumeric user interface

The alphanumeric user interface is a simple interface which enables you to debug Prolog programs even if you have a terminal without graphics capability. If you have a OSF/-Motif interface for your Prolog system, you can also debug your Prolog programs under the OSF/Motif graphical user interface. For further information, see the next section.

Unless you have redefined the input/output streams, the debugger reads the inputs from the keyboard (`keyboard`) and sends outputs to the screen (`screen`). For information on redefining streams, refer to chapter 5.

If you call the debugger for the predicate `ancestor/2`, the following screen is displayed:



The topmost two screen lines are used for status indication. The bottom four lines are reserved for command input, error messages and outputs to the standard output and standard error output. The lines in between form the output window, the length of which is variable. You can have different information fields displayed in the output window, and also scroll them if the information field is larger than the output window.

Status indicator

The status indicator displays the fully qualified designation of the current predicate and field-related notes, indicating for example which information field you are located in (`AND-BOX`). In the second line, the debugger displays information on explicit breakpoints or on pending constraints (`BEGIN`).

Command line

You enter debugger commands in the command line (`Command:`). The command line is the only screen line which you can write to. If the command which you enter is longer than

the command line, the debugger automatically continues writing in the message area. The admissibility of a command does not depend on the currently displayed field. The only exception is the command ⏎ . If the debugger is displaying the current box, ⏎ is the command for step-by-step operation. In any other field ⏎ is the command which causes the debugger to return to and display the current box.

Message lines

In the first message line the debugger displays error messages relating to the commands which have been issued. The other two lines display messages which are not generated by the debugger. If these outputs are longer than the two lines, the lines are scrolled up into the output window. At the next breakpoint the structure of the debugger screen is restored and only the last lines of the output are retained.

Output window

You can have the following information fields displayed in the output window:

- Help field
- Goal box
- Clause box
- Field for settings
- Field for breakpoints
- Field for old goal and clause boxes
- Field for ancestor goals
- Field for database
- Constraint field for variables
- Constraint field for subgoals
- Trace field

The information field presently visible in the output window is the current field. You can fetch the information fields for help, goal box, clause box, old goal and clause boxes and trace into the output window by means of commands. The other information fields appear only as a side effect of commands relating to the current field.

The following commands enable you to fetch the information fields into the output window:

Command	Short form	Meaning
help		Display help field in output window
view_and_box	&	Display goal box in output window
view_or_box	#	Display clause box in output window
view_previous_screen	<	Display previous goal or clause box
view_last_screen	<<	Display last goal or clause box
view_next_screen	>	Display next goal or clause box
view_first_screen	>>	Display first goal or clause box
view_history	vh	Display current goal or clause box
view_stops	vs	Display explicit breakpoints
view_trace	vt	Display trace field
`+`		Scroll forward line by line in the information field
`++`		Scroll forward page by page in the information field
`-`		Scroll back line by line in the information field
`--`		Scroll back page by page in the information field
snapshot	sn	Output current information field to a file

The information fields may be longer but not wider than the output window. In the information area the debugger displays `+` if the field extends beyond the lower window border and `-` if it extends beyond the upper window border. The commands `+, ++, -, --` enable you to scroll in the information fields. You do not need to switch to a special mode to do so.

The fields showing parts of the proof tree (`AND-BOX, OR-BOX`) are stored in a history list, the length of which you can change. The commands `<, <<, >, >>` enable you to scroll in these information fields. You do not need to switch to a special mode to do so.

Help field

In the help field you will find information on the debugger commands. You select the help field with the `help` command.

Field for AND-BOX

The current outermost port and the current subgoal appear in this field. If the clause does not fit into the output window, the debugger displays only part of it. But the commands `+,++,-,--` allow you to scroll in the output window. The debugger truncates overlong lines at the window border. The command `snapshot` allows you to store the screen contents in a file without restriction of line length, and to examine them further there.

The goal box becomes the current field when the debugger reaches a breakpoint at an outermost port. If this is an explicit breakpoint, the message `BREAKPOINT #` will appear in the status indicator, where # stands for the number of the breakpoint.

You select the AND-BOX with the command view_and_box or &.

Field for OR-BOX

The field for the OR-BOX shows the clause and its environment in the database. If the clause does not fit into the output window, the debugger displays only part of it. But the commands +,++,-,-- allow you to scroll in the output window. The debugger truncates overlong lines at the window border. The command snapshot allows you to store the screen contents in a file without restriction of line length, and to examine them further there.

The clause box becomes the current field when the debugger reaches a breakpoint at an internal port. If this is an explicit breakpoint, the message BREAKPOINT # will appear in the status indicator, where # stands for the number of the breakpoint.

You select the OR-BOX with the command view_or_box or #.

Field for settings

This field displays the current settings for the debugger and the interface, for example the current structure depth, or how many clause or goal boxes are to be stored.

You select the field for settings with the command view_configuration or vco.

Field for breakpoints

In this field the debugger displays the breakpoints which you have defined explicitly. The breakpoints are displayed as follows:

* *Number i Predicate Clause Subgoal Port Condition Actions*

* *Number i Predicate Clause Variable Instantiation Condition Actions*

The asterisk at the beginning of the line identifies an active breakpoint, i an interactive breakpoint. The number allows you to address those breakpoints which you wish to activate, deactivate or delete. Each breakpoint has such a *Number* which is automatically assigned to it by the debugger.

You select the field for breakpoints with the command view_stops or vs.

Field for old goal and clause boxes

You can view old goal and clause boxes in this field. You can set the number of boxes to be stored with the command set_history. The message PREVIOUS BOX # appears in the information area, where # specifies the chronological offset relative to the current goal or clause box.

You select the field for old goal or clause boxes with the commands <, <<, >, >> which simultaneously also enable you to scroll in this field.

Field for ancestor goals

In this field the debugger displays the dynamic ancestor goals for the current goal. You select the field for ancestor goals with the command `view_ancestors` or `va`.

Field for database

This field allows you to view a section in the database. The clauses for each predicate and the subgoals for each clause are consecutively numbered, so as to enable you to specify the numbers for setting explicit breakpoints.

You select the field for the database with the command `view_predicate` or `view_module`, or their short forms `v` and `vm`.

Constraint field for variables

The command `view_variable` or `vv` enables you to display all the defined constraints for a variable in the constraints field.

Constraint field for subgoals

If the next subgoal in the proof is a constraint goal, this is announced with `CONSTRAINT PENDING` in the status indicator. You can use the command `view_constraints` or `vc` to display in this field the position in the program at which processing is to continue. If no constraint goal is pending processing, an error message will be issued following the command `view_constraints`.

Trace field

If you have not defined a separate output predicate (`trace_output/4`) for trace or have not redirected the output stream, the trace outputs are displayed in a separate trace field. You can change the size of this field with the command `set_trace_length`.

You can select the trace field with the command `view_trace` or its short form `vt`.

Example

If you call the debugger for the goal `ancestor(`X`, pepsi)` from the example on page 166, you will get the following screen:

```
[user] ?- debug(ancestor(X, pepsi)).↩
```

```
------------------------------------------------------------
|call(ancestor(X,pepsi)) @ user                     AND-BOX |
-----------------------------------------------------BEGIN--
|debug :-                                                   |
| CALL>    call(ancestor(X,pepsi)) @ user.                  |
|                                                           |
|                                                           |
|                                                           |
|                                                           |
|                                                           |
|                                                           |
|                                                           |
|                                                           |
|                                                           |
|                                                           |
|                                                           |
|                                                           |
|                                                           |
|Command: v(ancestor/2) <ENTER>                             |
|                                                           |
|                                                           |
|                                                           |
|                                                           |
------------------------------------------------------------
```

The `AND-BOX` is displayed in the current output field; `BEGIN` signals that the debugger is situated at the first outer port of the box model. Then enter the command `v(ancestor/2)` in the command line.

```
-------------------------------------------------------------
|call(ancestor(X,pepsi)) @ user                     DATABASE|
-----------------------------------------------------BEGIN--
|1 ancestor(X,Y) :-                                         |
|   1            father(X,Y).                               |
|                                                           |
| 2 ancestor(X,Y) :-                                        |
|   1            mother(X,Y).                               |
|                                                           |
| 3 ancestor(X,Y) :-                                        |
|   1            father(X,Z),                               |
|   2            ancestor(Z,Y).                             |
|                                                           |
| 4 ancestor(X,Y) :-                                        |
|   1            mother(X,Z),                               |
|   2            ancestor(Z,Y).                             |
|                                                           |
|                                                           |
|                                                           |
|Command: stop_interactive([user:ancestor/2,1,1],call,[],[])<ENTER>
|                                                           |
|                                                           |
|                                                           |
|                                                           |
-------------------------------------------------------------
```

The field for the database is output in the output window, which displays that section of the database which defines the predicate `ancestor/2`. You now use the following command to define an interactive breakpoint and then have the field for breakpoints displayed in the output window.

```
stop_interactive([user:ancestor/2, 1, 1], call, [], [])

stop_interactive([user:ancestor/2, 1, 2], redo, [], [])
```

```
------------------------------------------------------------
|call(ancestor(X,pepsi)) @ user                       STOPS |
-----------------------------------------------------BEGIN--
|* 1   i  user : ancestor/2 1   1  call [] []               |
|* 2   i  user : ancestor/2 1   2  redo [] []               |
|                                                           |
|                                                           |
|                                                           |
|                                                           |
|                                                           |
|                                                           |
|                                                           |
|                                                           |
|                                                           |
|                                                           |
|                                                           |
|                                                           |
|                                                           |
|Command: <ENTER>                                           |
|                                                           |
|                                                           |
|                                                           |
|                                                           |
------------------------------------------------------------
```

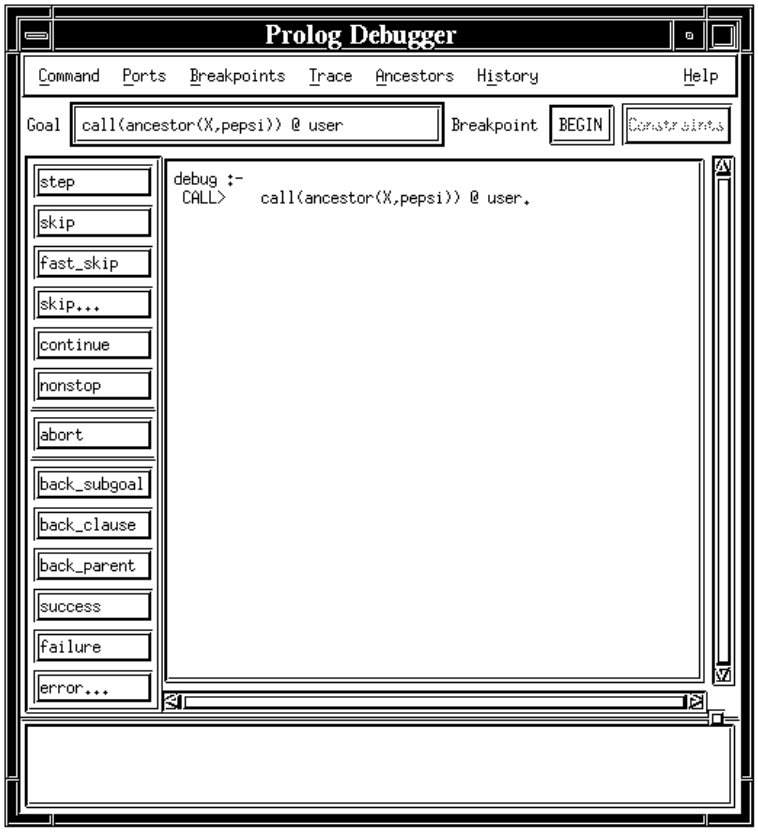
The command `Continue` or ↩ then produces the following output from the debugger.

```
-------------------------------------------------------------
|user : father(gregor,pepsi))                        AND-BOX |
-----------------------------------------------BREAKPOINT 1--
|ancestor(gregor,pepsi) :-                                   |
| CALL>    father(gregor,pepsi).                             |
|                                                            |
|                                                            |
|                                                            |
|                                                            |
|                                                            |
|                                                            |
|                                                            |
|                                                            |
|                                                            |
|                                                            |
|                                                            |
|                                                            |
|                                                            |
|Command:                                                    |
|                                                            |
|                                                            |
|                                                            |
|                                                            |
-------------------------------------------------------------
```

9.5 Debugging under the OSF/Motif user interface

The Prolog system provides a window-oriented debugger under OSF/Motif. The following section describes how you call the Prolog debugger under the OSF/Motif user interface, and use the menus, windows and buttons. If the Prolog system has been configured with OSF/Motif, the OSF/Motif user interface is loaded when the debugger is called.

If you call the debugger for the predicate `ancestor/2`, you will get the following screen:



Debugging under the OSF/Motif user interface

OSF/Motif opens a separate window for the Prolog debugger. In the title bar of the window you will find the typical OSF/Motif buttons for the window menu, minimize and maximize, and also the name of the application running in this window, i.e. *Prolog Debugger*.

Then you will see the menu bar with the entries `Command`, `Ports`, `Breakpoints`, `Trace`, `Ancestors` and `History`. Beneath the menu bar is the information bar which displays the current subgoal being debugged, and the breakpoint where current goal execution has halted. The `Constraints` button allows you to have pending constraints, if any, displayed in a separate window.

The Prolog output field, which displays the current section of the proof tree, occupies the major part of the screen. To the left of the output field is a row of buttons which execute the associated debugger command as soon as you click on a button. The Prolog system displays any exceptions in the bottom field of Prolog Debugger.

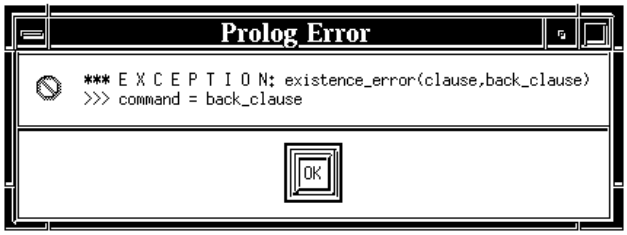
9.5.1 Controlling execution

Both the command buttons to the left of the output field and also the menu bar contain debugger commands. You will use the commands on the command buttons most frequently, those in the `Command` menu less often.

In the `Command` menu you will find the entries `break, halt, system, call, unify, interrupt, kill, command` and `quit`; these activate the corresponding debugger commands:

Menu entry	Meaning
break	New input loop
halt	Terminate Prolog session
system	Start operating system shell
call	Call Prolog goal
unify	Instantiate variable
interrupt	Interrupt current goal execution
kill	Terminate Prolog session
command	Enter and execute debugger command
quit	Exit debugger

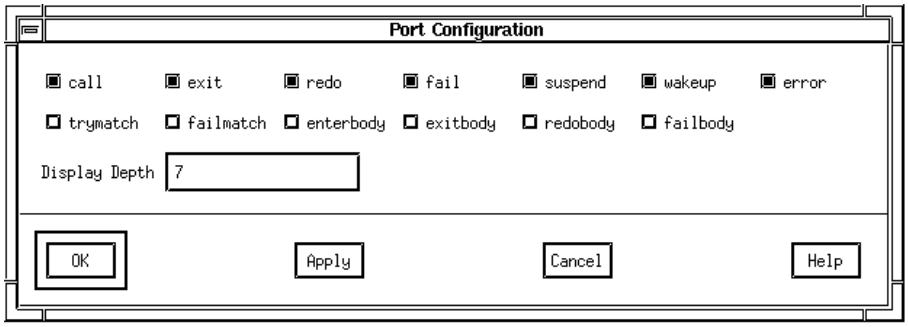
When you click on a menu item in the `Command` menu, the associated debugger command is executed. Some commands are followed by three periods, e.g. `error...`, which indicate that you must enter further parameters for the command in a separate window. If an error occurs here, for example if you enter an incorrect command, you will receive an error message in the form of the following dialog box. You should acknowledge this by clicking on the `OK` field. In addition, the exception is also output in the bottom field of the main window.



Error message window

9.5.2 Setting ports

The `Ports` menu allows you to change the default settings for the debugger and to set implicit breakpoints at the ports of the box model. The following diagram illustrates the default settings for the activated ports.



Setting implicit breakpoints

When you click on a button with the mouse pointer, you activate or deactivate an implicit breakpoint.

At the bottom of the window you will see four buttons which allow you to control operation of the dialog window:

`OK`
: The entered data is accepted, the command executed and the dialog window closed.

`Apply`
: The entered data is accepted and the command executed. The dialog window remains on the screen.

`Cancel`
: The command is not executed and the dialog window is closed.

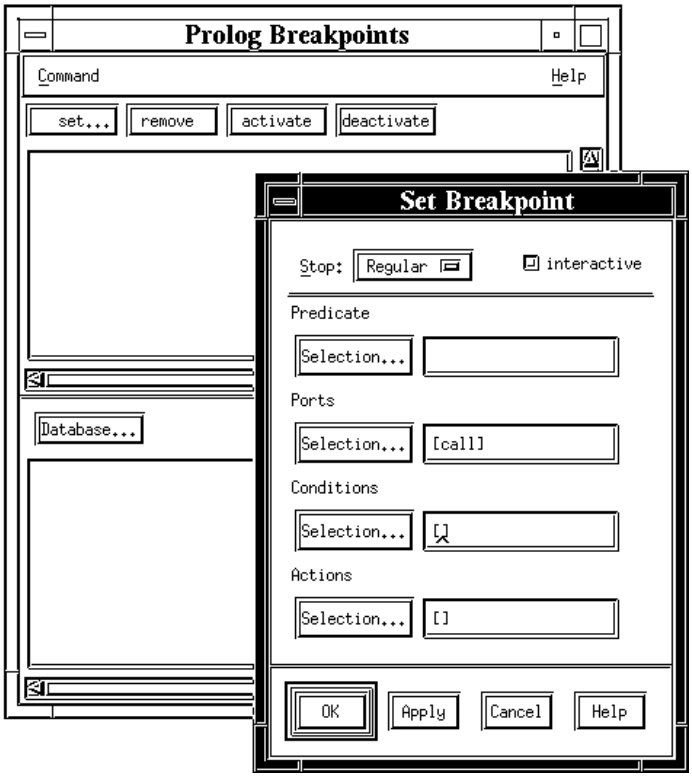
`Help`
: Help information for the dialog window is displayed in a separate window.

9.5.3 Setting explicit breakpoints

The `Breakpoints` menu allows you to set, delete, activate and deactivate explicit breakpoints (see also page 171), by clicking on the associated button.

The window for explicit breakpoints contains two text windows. The upper text window contains the breakpoints which you have set up, the lower one contains a section of the database with the program to be debugged. With the button `Database...` you activate a dialog window in which you can specify the section of the database.

If you click on the button `set...`, the `Set Breakpoint` window is opened on the screen, in which you can define the breakpoint.



Setting explicit breakpoints

You can use the top two buttons of this window to determine firstly whether a variable is to be monitored during goal execution or whether goal execution is to be regularly debugged (`Stop`). Secondly you can specify whether you wish to set an interactive or non-interactive breakpoint (`interactive`).

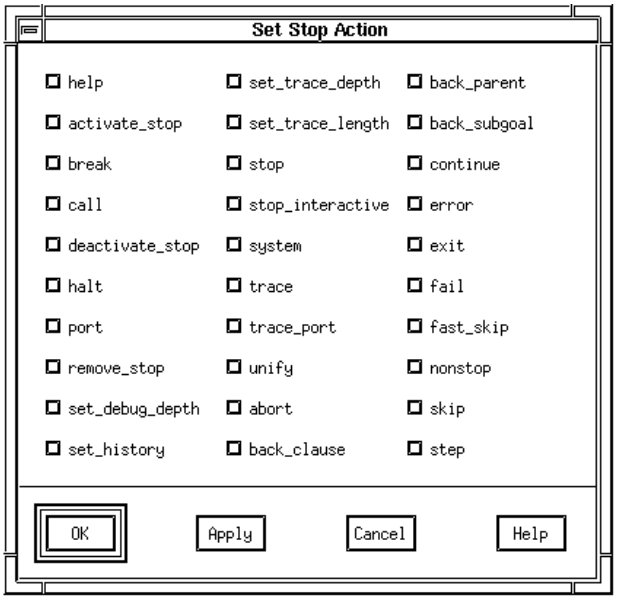
The button `Predicate Selection...` allows you to specify the predicate of the program for which you wish to set a breakpoint.

With the button `Ports Selection...` you determine the port at which the breakpoint is to be set.

With `Conditions Selection...` you can define the conditions (see also page 172) which must be satisfied before goal execution is interrupted at the breakpoint.

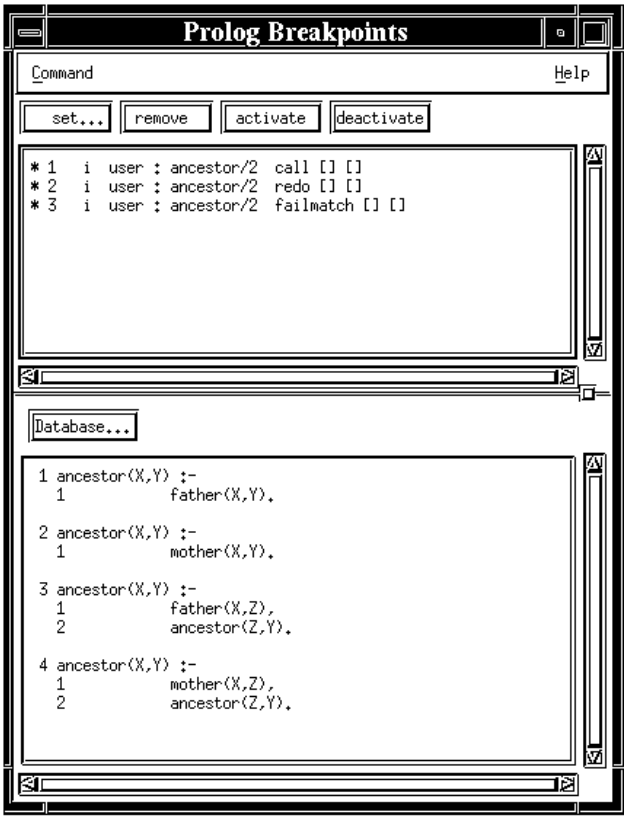
Finally, `Actions Selections...` allows you to define all the actions which are to be performed after stopping at the breakpoint.

All `Selections...` buttons open a separate window in which you make relevant entries.



Possible actions at an explicit breakpoint

If you debug the example from page 166 under the OSF/Motif user interface, the window `Prolog Breakpoints` will contain the following entries:

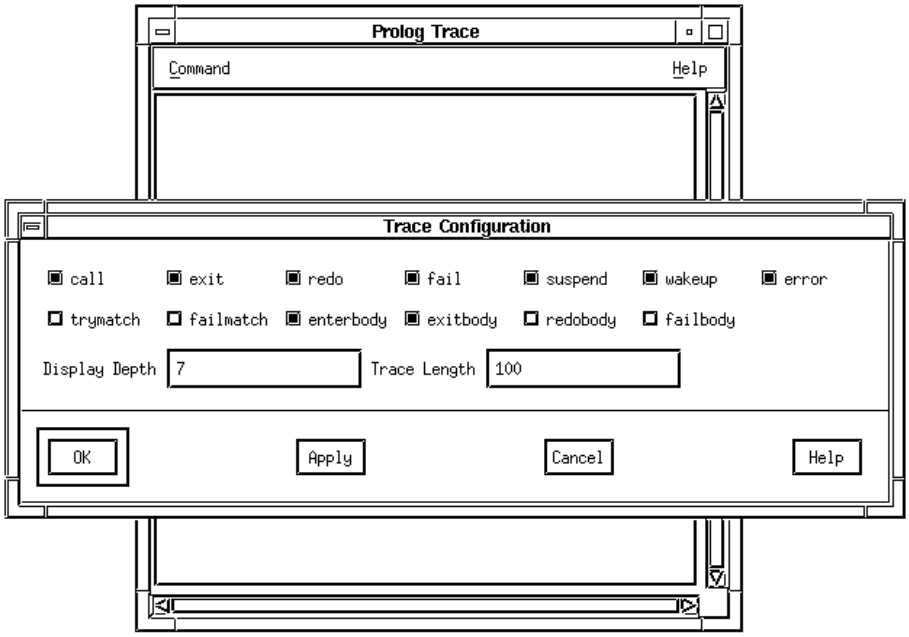


Predicate ancestor/2 with 3 interactive breakpoints

9.5.4 The Tracer

In the `Trace` menu you can globally turn on or off the trace, or activate it for an individual subgoal. If you wish to take a look at the trace output, click on the menu item `open...`.

This opens the window **Prolog Trace**. In the `Command` menu for this window you can use the menu command `configure` to define the ports to be logged when they are passed through.

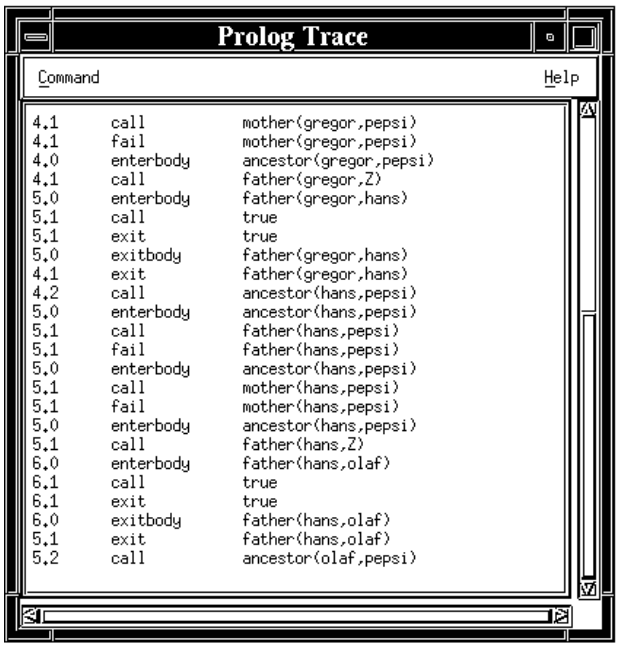


Configuring trace

The menu command `clear` allows you to clear all the outputs displayed in the output field of the trace window. The menu command `quit` closes the trace window.

Example

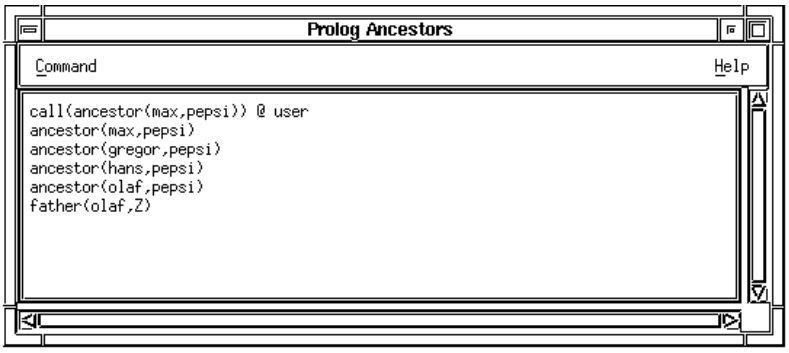
The following diagram shows the trace outputs for the predicate `ancestor(`X`,pepsi)` from the example on page 166.



Trace outputs

9.5.5 Displaying ancestors

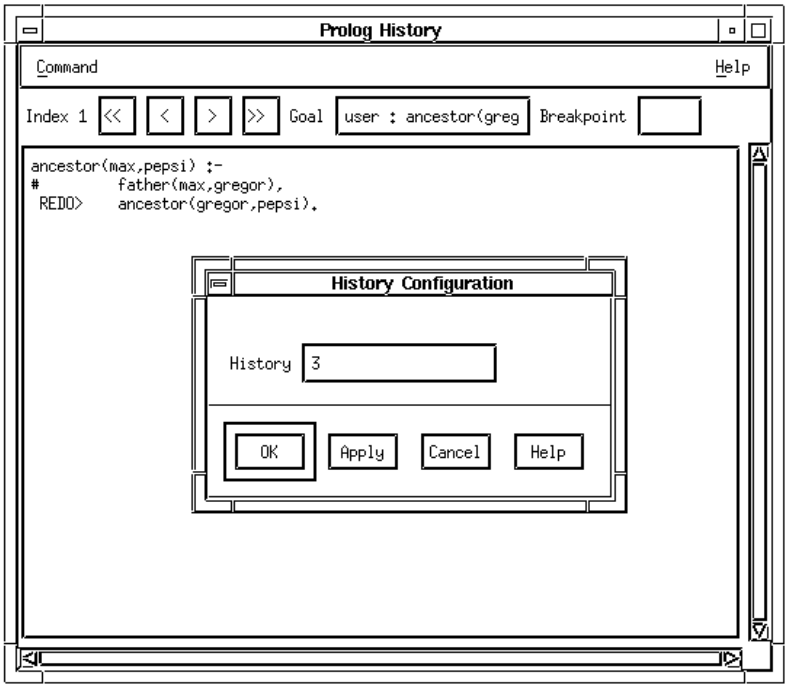
The menu **Ancestors** activates a window in which the dynamic chain of Prolog goals is displayed. The contents of this window are not automatically refreshed, but only updated with new information whenever the menu item is activated.



Dynamic call chain

9.5.6 Debugger log

The contents of the text fields in the **Prolog Debugger** window are stored internally by the debugger. The **History** menu allows you to take another look at the most recent outputs.



Configuring the debugger log

In the **Command** menu of the History window, the menu item `configure` lets you configure the debugger. The number you specify determines how many sections of the proof tree are stored.

The buttons $<<, <, >, >>$ allow you to display the first, the previous, the next or the last section of the proof tree. `Index` indicates which section in the series of stored section is currently being displayed.

9.6 Summary tables of debugger commands

The debugger commands, ordered by function, are summarized in the following tables. The commands are described individually in the *IF/Prolog Reference Manual* [1].

Requesting information

Command	Short form	Meaning
help	h	Display help information
trace	t	Log passage through ports
view_ancestors	a	Display dynamic call chain with all ancestors
view_stops	vs	Display list of all explicit breakpoints
view_variable	vv	Display all constraints for a variable
view_constraints	vc	Display all constraints
view_configuration	vco	Display all debugger settings
view_predicate	v	Display predicate
view_module	vm	Display all predicates for a module
view_and_box	&	Display goal box in output window
view_or_box	#	Display clause box in output window
view_previous_screen	<	Display previous goal or clause box
view_last_screen	<<	Display last goal or clause box
view_next_screen	>	Display next goal or clause box
view_first_screen	>>	Display first goal or clause box
view_history	vh	Display current goal or clause box
view_stops	vs	Display explicit breakpoints
view_trace	vt	Display trace field
+		Scroll forward line by line in the information field
++		Scroll forward page by page in the information field
-		Scroll back line by line in the information field
--		Scroll back page by page in the information field
snapshot	sn	Output current information field to a file

Setting the configuration

Command	Short form	Meaning
port	p	Turn port monitoring on/off
trace_port	tp	Turn port monitoring on/off in the trace
set_debug_depth	sdd	Set structure depth
set_trace_depth	std	Set structure depth
set_trace_length	stl	Set length of trace field
set_history	sh	Set length of history field
trace	t	Turn trace on/off

Controlling the debugger

Command	Short form	Meaning
abort	A	Terminate interactive debugger
break	B	Initialize new input loop
halt	H	Terminate Prolog session
system	S	Start shell
call	ca	Call Prolog goal

Working with breakpoints

Command	Short form	Meaning
remove_stop	rs	Delete explicit breakpoint
stop	st	Set explicit breakpoint
stop_interactive	si	Set interactive breakpoint
deactivate_stop	ds	Deactivate breakpoint
activate_stop	as	Reactivate breakpoint

Manipulating the database

Command	Short form	Meaning
unify	=	Instantiate variable

Step-by-step processing

Command	Short form	Meaning
step	↩	Process to next monitored port
skip	s	Skip monitoring of current subgoal
fast_skip	fs	Fast-skip monitoring of current subgoal
continue	c	Execute to next breakpoint
nonstop	no	Execute to end of proof

Branching in the proof tree

Command	**Short form**	**Meaning**
back_subgoal	b	Branch back to `call` port of subgoal
back_parent	bp	Branch back to `call` port of parent goal
back_clause	bc	Branch back to clause head
fail	f	Force failure of current subgoal
exit	x	Force success of current subgoal
error	err	Generate exception

Chapter 10

The C interface

IF/Prolog provides two paths to the programming language C. Using the active C interface you can implement predicates as C functions and then use them like normal predicates in cases where

- a problem can be solved only in C
- large application packages are already available for you to use
- the solution in C is more efficient than the one in Prolog.

Possible applications include:

- access to the operating system
- interfaces to other programming languages such as FORTRAN, Pascal etc.
- access to an existing database system

Using the passive C interface you can call Prolog predicates from a C program just like C functions and, if necessary, initiate backtracking. It is thus a flexible and easy-to-use facility for combining C functions and Prolog predicates, and might be used in contexts where

- C programs can use the Prolog database like a relational database;
- C programs can access Prolog programs such as expert systems;
- existing C programs can be extended by means of rapid prototyping with Prolog predicates;
- applications which are difficult to program in C, such as those requiring backtracking, can be written in Prolog and then integrated into a C program.

You can also call functions of the passive C interface recursively and mix them with functions of the active C interface.

The first section of this chapter provides an overview of the active C interface. It explains how you can write simple C functions, i.e. functions which do not allow backtracking. Predicates allowing backtracking can be implemented by means of complex C functions which are described in section 10.1.2.

The section 10.2 gives you a short overview of the passive C interface with tables summarizing the data types, macros and functions of this interface. Then you will find a detailed description of how to call a Prolog predicate by means of a C function. Examples are also provided.

The *IF/Prolog Reference Manual* [1] contains descriptions, in alphabetical order, of the built-in C functions of the C interface.

The section 10.3 of this chapter describes how to implement a C device driver.

The function `Cboot()` provides you the method of introducing your code to all three C interfaces to the Prolog system. You can use the skeleton makefile *model_make* to compile and link your code with the Prolog system. For information on the steps you must perform to link C and Prolog, see page 238.

10.1 Active C interface

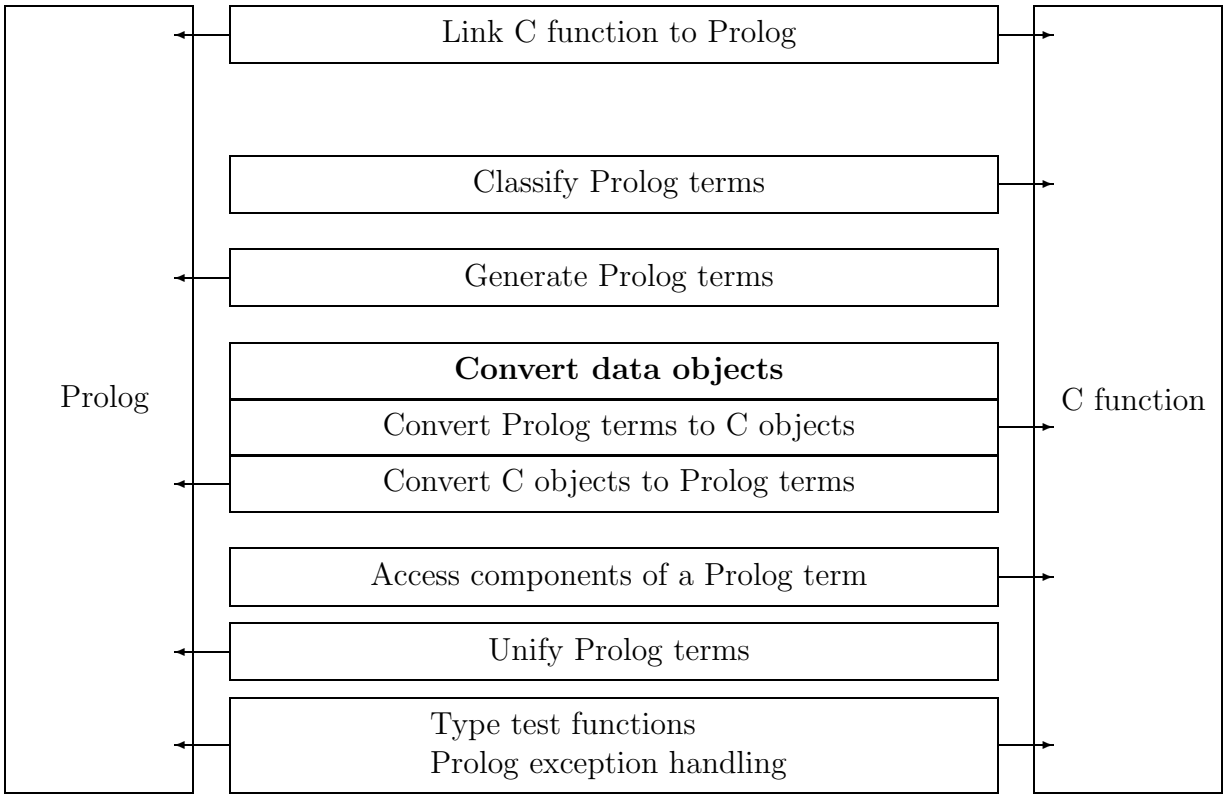
There are two classes of C functions which can be used in Prolog using the active C interface:

- simple C functions are functions which can be used in Prolog as predicates without backtracking. You will find a description of how to write such functions on beginning page 202.
- complex C functions
  are incorporated as Prolog predicates with backtracking. The backtracking concept has to be programmed into the execution sequence of the C function. The manner in which this is done is defined by the active C interface. The conventions which apply are explained in section 10.1.2.

The active C interface is designed to reconcile the following differences between the language concepts of C and Prolog:

- Distinct data objects
  The data objects in Prolog (terms) are different from the data types in C (`int`, `double`, `char`, `...`).
- Unification and value assignment
  The unification of terms and the associated instantiation of variables in Prolog differs significantly from the way in which a value is assigned to a C variable.

To resolve these differences, the active C interface provides a set of built-in functions. The following diagram presents an overview of the areas covered by these functions. In each case the arrow indicates the direction of information flow.



Functions of the active C interface

Overview of the built-in data objects and functions

This section summarizes the library functions of the active C interface, grouped according to the tasks which they perform. All built-in data objects and functions are declared in the include file *cpred.h* which you will find in the directory *$PROROOT/c*.

Data objects of the C interface

All data objects in Prolog are terms. Internally, every Prolog term is represented by a machine word containing an appropriate identifier for the class to which the term belongs. The data type name `TERM` is defined for this data type in the include file *cpred.h*. This data type name is used to declare C variables which can accommodate Prolog terms and are referred to as term variables in the following. Term variables are objects which are administered by the C interface. They should never be manipulated directly.

Prolog objects	Representation in C	Data type name
Term	Implicit type conversion	TERM

The following facilities of C are used for mapping Prolog terms onto C objects:

C objects	Representation in C	Data type name
Integer	`long`	`long`
Floating-point number	`double`	`double`
String	C character string, i.e. an array of characters terminated by a null byte `'\0'`	`STRING`

The data type names `ARITY` for the arity of structures and `STRING` for strings or names are also declared in the include file *cpred.h*. The use of these names is recommended in order to avoid inconsistent data types. If you use data types other than `signed long` for integers and `double` for floating-point numbers in a C function, you must insert appropriate cast operators before a type conversion to a Prolog term is performed. You should use only the built-in C functions of the active C interface for the conversion, as inconsistencies in the internal representation may otherwise result.

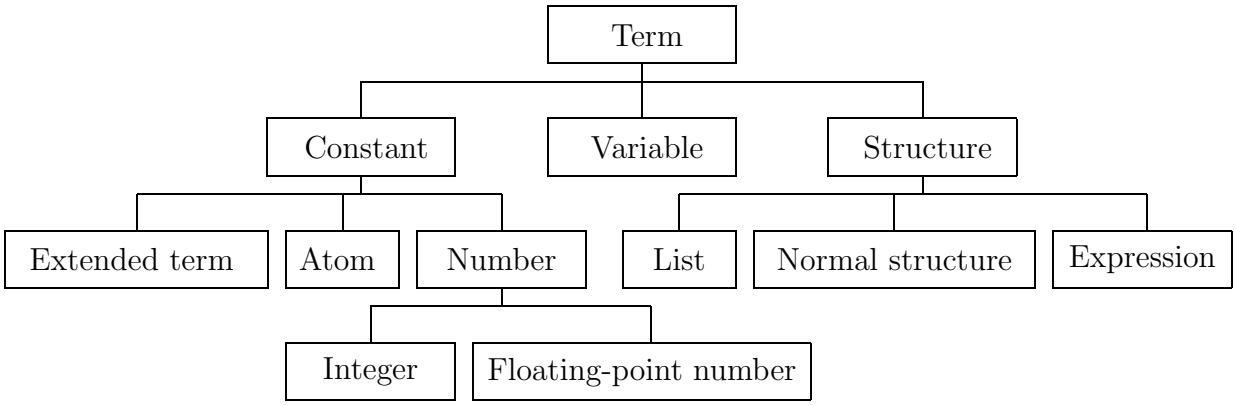
Memory for term variables

The C interface manages all the terms which you can access using term variables. You must declare the term variables before calling built-in functions which write terms to term variables. The active C interface itself takes care of any memory expansion or garbage collection. It retains all term variables until the C predicate creating them has been terminated or until the function itself explicitly release the memory.

C function	Purpose
TermCollect()	Release memory for specific term variables
TermCollectSequence()	Release memory for specific term variables
TermContext()	Determine current term context

Classifying Prolog terms

The basic data objects of Prolog are terms (see the chapter on *Syntax* in the *IF/Prolog Reference Manual* [1]). The following classification hierarchy applies:



The active C interface offers various C functions which allow you to classify a term more precisely and, if this is successful, to convert it into a C data type. In the event of an error, these functions generate certain exceptions.

C function	Purpose
TermIsAtom()	Check atom and convert to string
TermIsCompound()	Check and convert structure
TermIsFunctor()	Check and convert structure
TermIsInteger()	Check and convert integer
TermIsIntegerExpression()	Check and evaluate integer expression
TermIsList()	Check and convert list
TermIsNil()	Check empty list
TermIsFloat()	Check and convert floating-point number
TermIsFloatExpression()	Check and evaluate arithmetic expression
TermIsUniversal()	Check and convert structure
TermIsVar()	Check variable

Analyzing Prolog terms

You can also classify a term by using the functions `TermType()` and `TermDecompose()`.

C function	Purpose
TermDecompose()	Analyze term and query additional information
TermType()	Analyze term

Accessing components of a term

C function	Purpose
PrologArg()	Access an argument of the active C predicate
TermArg()	Access an argument of a Prolog structure

Generating Prolog lists in C

Lists are terms which are used very often in Prolog. The active C interface therefore supports the processing of lists with special functions.

C function	Purpose
TermOpenList()	Generate list
TermAddList()	Add element to list
TermCloseList()	Close list

Generating Prolog objects

The following functions allow you to convert C values into Prolog objects:

C function	Purpose
TermMakeAtom()	Generate Prolog atom
TermMakeCompound()	Generate Prolog structure from functor, arity and arguments
TermMakeFunctor()	Generate Prolog structure from functor and arity
TermMakeInteger()	Generate Prolog integer
TermMakeList()	Generate Prolog list
TermMakeNil()	Generate empty Prolog list
TermMakeFloat()	Generate Prolog floating-point number
TermMakeUniversal()	Generate Prolog structure with uninstantiated variables
TermMakeVar()	Generate uninstantiated Prolog variable

Unification

The unification functions allow you, after converting C values into terms, to return them to Prolog.

C function	Purpose
TermUnifyAtom()	Unify Prolog atom
TermUnifyCompound()	Unify Prolog structure
TermUnifyFunctor()	Unify Prolog structure
TermUnifyInteger()	Unify Prolog integer
TermUnifyList()	Unify Prolog list
TermUnifyNil()	Unify empty Prolog list
TermUnifyFloat()	Unify Prolog floating-point number
TermUnifyUniversal()	Unify Prolog structure
TermUnifyVar()	Unify uninstantiated Prolog variable

Check functions and exception handling

An error context is a particular set of data which the Prolog system requires in order to generate an exception (see also chapter 8). Before a C predicate is called, Prolog clears the error context. If a C predicate fails, Prolog checks the error context. If this is set, Prolog initiates exception handling (`exception/2`).

C function	Purpose
ErrorContextClear()	Clear error context
ErrorContextIsSet()	Check whether an error context is set
ErrorContextGet()	Query error context
ErrorContextSet()	Set error context

i The term classification functions `TermIs...()` set the error context implicitly in the event of an error.

10.1.1 Linking simple C functions

Simple C functions are those C functions which, when used as a predicate, do not initiate backtracking. First, the general procedure for including a simple C function in Prolog is explained step by step. The procedure is then applied to an example. The various built-in C functions used in the example are described in detail in the *IF/Prolog Reference Manual* [1].

- Programming a function
  First, write the C function *c_function* and store the source text for the function in a file *file.c*.
- Write the include statement

  ```
  #include "cpred.h"
  ```

  as the first line of file *file.c*. The include file *cpred.h* contains macros and type definitions required for incorporating the C function.
- After this include statement, insert a `Cboot()` function to your file *file.c*, based on the following model:

  ```
  void Cboot(void)
  {
          CPRIM("functor", arity, c_function);
          .
          .
  }
  ```

  In this way, you set up a link between Prolog and the *c_function* and assign the predicate with *functor* and *arity* to the C function.

i You may define only one `Cboot()` function. For each simple C function which you wish to incorporate, `Cboot()` must include precisely one `CPRIM()` or `MCPRIM()` call with appropriate arguments.

- Passing parameters between C and Prolog
  The function `PrologArg(n)` gives you the n-th argument of the newly defined predicate. By using the functions of the function class `TermUnify...()` you can unify the results of the C function with the arguments of the predicate and thus pass them to Prolog. To perform type checking and conversion to C variables you use the functions of the function class `TermIs...()`.

- Returning `TRUE` or `FALSE` as result
  You should ensure that your function yields a result using the return statement, as follows:

Return value	Interpretation in C	Goal executing
TRUE	1	Successful
FALSE	0	Unsuccessful

  If the return value is `FALSE`, the Prolog system automatically undoes any variable instantiations produced by `TermUnify...()`.

- Generating exceptions
  If you want to set an exception so that Prolog exception handling will be invoked in the event of fail, you must use the C function `ErrorContextSet()`. However, you should note that the functions of the function class `TermIs...()` set the error context implicitly.

- Generating an extended Prolog system
  In order to include your C function definitively in Prolog as a new predicate, you need to generate an extended Prolog system (see section 10.4).
  In our case the extended Prolog system has been given the name *myprolog*, but you are free to choose any other name.

- Debugging in Prolog
  To call the extended system you have just generated, you enter the command

  $ `myprolog` ↩

  As soon as your new Prolog system displays the prompt, you can test your new predicate.

Example

In this example we want to program a C function which will accept a character string as a parameter and yield the length of the character string as its result. The function is then to be incorporated in Prolog as a new predicate.

- Programming the function

  The core component here is the function *len()*

```
int len(const char *text, long *length)
{
        *length = strlen(text);
        return 1;
}
```

- Testing the function

  The following main function allows you to test the function *len()*

```
#include <stdio.h>
#include <string.h>

int main(int argc, char *argv[])
{
        long length;
        if (argc == 0)
        {
                fprintf(stderr, "At least one argument expected\n");
                return 1;
        }
        len(argv[1], &length);
        printf("Length = %ld\n", length);
        return 0;
}
```

The completed source file has the following contents:

```
#include <stdio.h>                                          /* 1 */
#include <string.h>
#include "cpred.h"                                          /* 2 */

static int len(const char *text, long *length)              /* 3 */
{
        *length = strlen(text);                             /* 4 */
        return 1;                                           /* 5 */
}

static BOOLEAN len_call(void)                               /* 6 */
{
        STRING text;                                        /* 7 */
        long length;                                        /* 8 */

        return  TermIsAtom(PrologArg(1), &text)             /* 9 */
```

```
        &&      len(text, &length) == 1                         /* 10 */
        &&      TermUnifyInteger(PrologArg(2), length);         /* 11 */
}

void Cboot(void)                                                /* 12 */
{
        CPRIM("len_pred", 2, len_call);                         /* 13 */
}
```

This achieves the following:

- Preparation for integration into Prolog
  The main function has been removed and the include statement inserted. The functor `len_pred` and the arity 2 are declared for the new predicate.
  A link is established between Prolog and the function *len()* by means of the function *len_call()* (line 6). This function handles the tasks arising from the succeeding steps.

- Parameter passing between C and Prolog
  `TermIsAtom()` checks whether the first argument is an atom; if so, it converts the atom into a C string. This is passed as the first argument to the function *len()* (line 10).
  `TermUnifyInteger()` converts the value of *length* in C representation into Prolog representation and unifies the result with the second argument of the predicate (line 11).

- Returning `TRUE` or `FALSE` as result
  All goals are proved only if the three functions `TermisAtom()`, *len()* and `TermUnifyInteger()` all yield the result `TRUE` (non-zero), i.e. if

  – the first argument when the predicate was called was really an atom,
  – the `len()` call was successful, and
  – the length of the atom could be unified with the second argument of the predicate.

  In all other cases the goal fails because at least one of the three functions yields the result 0 or `FALSE`.

- Exception handling
  If the first argument of the goal is not an atom, the function `TermIsAtom()` automatically sets the error context and generates an exception (line 9).

- Generating and testing an extended Prolog system
  Once you have generated and called the extended Prolog system as described in section 10.4, you can test the new predicate:

```
[user] ?- len_pred(hello, L).↩

L       = 5 ↩

yes
```

10.1.2 Complex C functions

Basically, the following differences exist between predicates with backtracking and C functions:

Predicate with backtracking	**C function**
Yields different solutions in the course of backtracking	Always yields the same solution when the same call arguments are used
Stores information regarding the point reached during the last activation of the predicate	Following termination of the function, all local information is lost

Because of these differences, simple C functions are not suitable for implementation of predicates with backtracking.

Predicates with backtracking may be implemented in a C context using the concept of **coroutines**. In order to clarify how this works, the following sections adopt a step-by-step approach to show how

- predicates with backtracking can be mapped onto the box model,
- the box model can be simulated using coroutines,
- coroutines can be implemented in C,
- a coroutine is linked to Prolog as a predicate with backtracking.

This completes the description of the conventions of the active C interface which govern how a complex C function is implemented as a predicate with backtracking.

A presentation of various macros which make it easier for you to incorporate certain coroutine operations correctly into your C function is also included. These macros enable frameworks to be built for two types of complex C functions, namely

- complex C functions without epilog
- complex C functions with epilog

In an **epilog**, you can define actions to be taken when an exit from a predicate is made through the `fail` port. A description of these frameworks and how they are used to program complex C functions is given on page 215; in each case their use is illustrated by means of an example.

The box model and coroutines

A Prolog predicate can be represented as a box with four ports (see also chapter 3). The following scenario is typical for a predicate with backtracking:

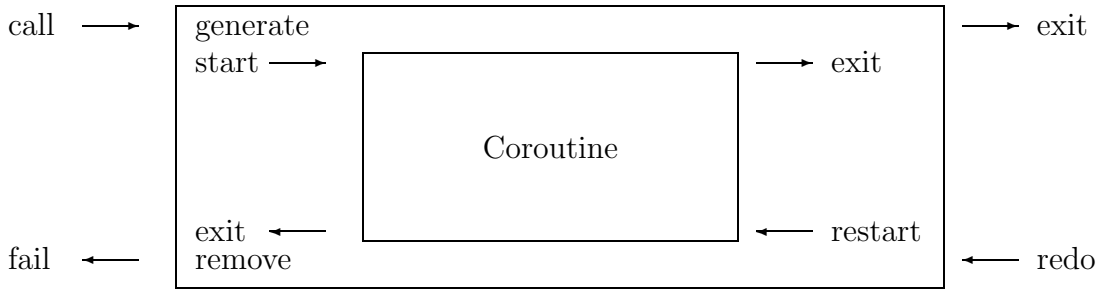
- The box is exited via the `exit` port and is not removed. All internal information within the box remains intact.
- The box is reentered via the `redo` port. The search for a solution is resumed from the point at which it was previously interrupted.

This sequence may be repeated several times. The box model itself can be simulated by a coroutine. A **coroutine** is a subprogram with the following special characteristics:

- it can be exited at a specific point without its internal information being lost, and
- it can be reentered as though it had never been exited: all variables and the control structures are exactly as they were when the exit was made.

This process can be repeated more than once at different points.

These similarities with the characteristics of the box model already suggest how the simulation will look. Coroutine operations are assigned to the box ports as follows:



`call` ⟷ generate and start A coroutine instance is generated, initialized and started, i.e. the search for the first solution is initiated.

`exit` ⟷ exit The coroutine instance is exited but not removed, i.e. local variables, processing position, etc. are stored so that the same instance can be executed again later.

`redo` ⟷ restart The variable instantiations made during the previous execution of the instance are undone. The instance is executed again; the processing position and local variables are taken from the previously stored values.

`fail` ⟷ exit and remove The coroutine instance is exited and removed.

Simulating a coroutine in C

In C, you must simulate the coroutine concept by means of C functions. You require the following for this purpose:

- a function with result type `int`
  This function handles the actual tasks involved. The central component of this function is a switch statement, which branches as necessary so that the correct entry point is reached at a start or restart.

- definition of a state structure
  The local variables of a coroutine instance are stored outside the function, in the state structure, so that they are not lost when the coroutine is exited.

- an `int` variable *PP* (program pointer)
  The program pointer contains information about the entry point for the start or restart.

Example

```
typedef struct { int x; } NameState;

int NameFunc(int PP, void *state)         /* start: PP = 0    */
{
        NameState *StatePtr = (NameState *) state;

        switch (PP)
        {
            case 0: break;
            case 1: goto Label1;
            case 2: goto Label2;
            case 3: goto Label3;
            default: return 0;
        }
        StatePtr->x = 1;                  /* <--------- start */
        printf("%d  : ",StatePtr->x);
        printf("Start\n");
        return 1;                         /* --> exit         */
    Label1:;                              /* <-- restart      */
        StatePtr->x ++                    /* PP = 1           */
        printf("%d  : ",StatePtr->x);
        printf("Redo\n");
        return 2;                         /* --> exit         */
    Label2:;                              /* <-- restart      */
        StatePtr->x ++                    /* PP = 2           */
        printf("%d  : ",StatePtr->x);
        printf("Last visit\n");
        return 3;                         /* --> exit         */
    Label3:;                              /* <-- restart      */
        return 0;                         /* PP = 3           */
                                          /* -------------->  */
```

```
                                                /* exit and remove  */
    }
```

This example offers an easy-to-follow guide to the general functioning of the concept:

- generate and start:
  The function is called with the program pointer PP = 0; a pointer to the state structure is passed as the second argument. The function uses the components of this state structure as local variables.
- exit without removal:
  The function returns a positive value. This is the new program pointer, it is stored outside of the function. The state structure remains intact.
- restart:
  The function is called with the most recently stored value of the program pointer PP > 0. It branches to the corresponding label. The local variables from the preceding call are still present in the state structure.
- exit and remove:
  The function returns a value of 0. The memory space used for the state structure and program pointer is released.

The first time the above function is called as a coroutine, it produces the output:

```
1 :  Start
```

A count of the number of calls is now recorded in component x of the state structure *NameState*. The first restart of the function produces the output

```
2 :  Redo
```

The next restart produces

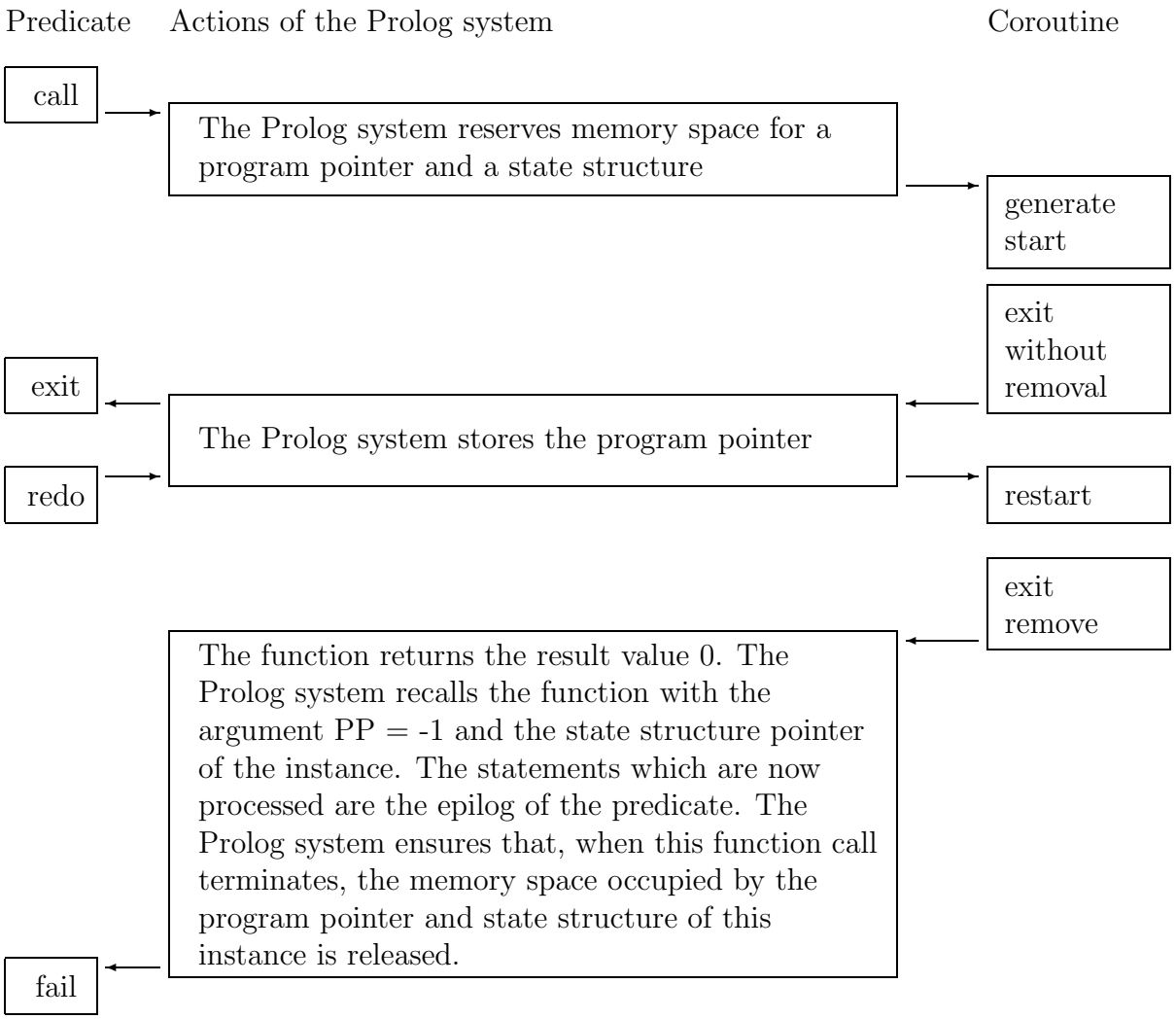
```
3 :  Last visit
```

On the next call, the instance of the coroutine is removed.

Linking a coroutine to Prolog

The active C interface for Prolog governs the final details for linking a coroutine. The link is set up by a call of the function `CPRED()` or `MCPRED()` in the `Cboot()` function. The call arguments inform the Prolog system of the functor of the new predicate, its arity, the name of the C function, and the size of the state structure.

Further actions by the system when the new predicate is called are shown in the following diagram. The arrows indicate the direction of execution flow.


Predicate
Actions of the Prolog system
Coroutine
call
The Prolog system reserves memory space for a program pointer and a state structure
generate start
exit without removal
exit
The Prolog system stores the program pointer
redo
restart
exit remove
The function returns the result value 0. The Prolog system recalls the function with the argument PP = -1 and the state structure pointer of the instance. The statements which are now processed are the epilog of the predicate. The Prolog system ensures that, when this function call terminates, the memory space occupied by the program pointer and state structure of this instance is released.
fail

Example

The simple coroutine is now to be expanded so that it can be linked to Prolog as a predicate with backtracking. During backtracking, the predicate `pred/2` is to unify its first argument successively with the integers 1, 2 and 3, and the second with the atoms `'Start'`, `'Redo'` and `'Last visit'`.

```
Source code for the complex                    | Corresponding
C function                                     | Prolog predicate
                                               | with clauses


#include "cpred.h"                             |
```

```
                                                    |
typedef struct { int x; } NameState;                |
                                                    |
int NameFunc(int PP, void *state)                   |
{                                                   |
   NameState *StatePtr = (NameState *) state;       |
   switch(PP)                                       |
   {                                                |
      case 0: break;                                |
      case 1: goto Label1;                          |
      case 2: goto Label2;                          |
      case 3: goto Label3;                          |
      default: return 0;                            |
   }                                                |
   StatePtr->x = 1;                                 | pred(X,Y):-
   if(TermUnifyInteger(PrologArg(1),StatePtr->x)    |   X=1,
   && TermUnifyAtom(PrologArg(2),"Start"))          |   Y='Start'.
      return 1;                                     |
   PrologUndo();                                    |
Label1:                                             |
   StatePtr->x ++;                                  | pred(X,Y):-
   if(TermUnifyInteger(PrologArg(1),StatePtr->x)    |   X=2,
   && TermUnifyAtom(PrologArg(2),"Redo"))           |   Y='Redo'.
      return 2;                                     |
   PrologUndo();                                    |
Label2:                                             |
   StatePtr->x ++;                                  | pred(X,Y):-
   if(TermUnifyInteger(PrologArg(1),StatePtr->x)    |   X=3,
   && TermUnifyAtom(PrologArg(2),"Last visit"))     |   Y='Last visit'.
      return 3;                                     |
   PrologUndo();                                    |
Label3:                                             |
   return 0;                                        | pred(X,Y):-
}                                                   |   fail.
                                                    |
void Cboot(void)                                    |
{                                                   |
  CPRED("pred",2,NameFunc, sizeof(NameState));      |
}                                                   |
```

If you write the source code into a file *file.c* and then generate an extended Prolog system, the new predicate `pred/2` is made available to you. This predicate now has the same properties as if you had defined it with the clauses given on the right in the above diagram.

Before a complex C function can be linked to Prolog, it has to be fitted, as in the example, into a framework with the following structure:

```
#include "cpred.h"

typedef struct { int x; int y; ... } State;

int Func_name(int PP, void *state)
{
        State *StatePtr = (State *)state;
        switch(PP)
        {
                case 0: break;
                case 1: goto Label1;
                case 2: goto Label2;
                     .
                     .
                default: return 0;
        }
            .
            .
        StatePtr->y = ... ;
        StatePtr->x = ... ;
            .
            .
        ... return 1;
        PrologUndo();
        Label1:
            .
            .
        ... return 2;
        PrologUndo();
        Label2:
            .
            .
        return 0;
}

void Cboot(void)
{
        CPRED("Pred_name", Arity, Func_name, sizeof(State));
}
```

10.1.3 Built-in macros and functions for connecting complex C functions

The macros and functions described below facilitate the development and linking of complex C functions. The macros and functions are listed here in the same logical order in which

they should be used in the program text. In the case of macros, the definition in the include file *cpred.h* is also given at the end of the macro description.

COROUTINE(c-function,State,StatePtr)

This macro call constitutes the beginning of the function.

c-function	Name of the new C function; this must be the same as the name specified in `CPRED()` or `MCPRED()`.
State	Type of a state structure defined in a typedef statement.
StatePtr	Pointer to a structure of type *State*.

bf Source code:

```
#define COROUTINE(n,t,s) \
        int n (int _p, void *_s) { t *s = (t *)_s; \
        switch ( _p ) { case 0: break;
```

RESUME(ProgPtr,Label)

specifies the point of restart following a re-entry. The parameters must match those given in the `DETACH` call.

ProgPtr	Number ($\neq$ -1) belonging to *Label*, at which restart is to occur.
Label	Name of the point of restart.

Source code:

```
#define RESUME(no, label) case no:  goto label
```

HASEPILOG

announces that the function has an epilog which can be activated if necessary.

Source code:

```
#define HASEPILOG case -1:  goto _epilog
```

BEGIN

comes immediately before the actual program code.

Source code:

```
#define BEGIN default:  FAIL; } {
```

void PrologEnableEpilog(void);

This function must be called immediately after `BEGIN` if the complex C function has an epilog (`HASEPILOG`).

DETACH(*ProgPtr,Label*)

The function is exited. *ProgPtr* is the return value. A new entry point is defined. The parameters must match those given in the RESUME call.

ProgPtr	Number ($\neq$ -1) belonging to *Label*, at which restart is to occur.
Label	Name of the point of exit from the function.

Source code:

```
#define DETACH(no, label) { return no; label:  ; }
```

void PrologUndo(void);

Unifications in the current call are undone.

FAIL

The coroutine instance is removed. It may be necessary for you to include an epilog to ensure that possible side effects are correctly handled.

Source code:

```
#define FAIL return 0
```

EPILOG

Defines a label to which a branch will be made if the coroutine instance is removed. The section of the program to be executed as an epilog (following the instance's removal) must be placed directly between `EPILOG` and `END`.

Source code: `#define EPILOG FAIL; _epilog`

END

Must be called as the last macro in the complex C function. The `END` call ensures that the function returns `FALSE` (the predicate fails) if it was not previously exited using `DETACH`.

Source code:

```
#define END } FAIL; }
```

Example

The macros and functions described above can be used to construct the following framework for facilitating error-free production of the source text for a complex C function without epilog.

```
#include "cpred.h"

typedef struct { ... } State;

COROUTINE(Funcname, State, StatePtr)
        RESUME(ProgPtr1, Label1);
        RESUME(ProgPtr2, Label2);
            .
            .
            .
        RESUME(ProgPtrN, LabelN);

BEGIN
        /* Start of actual program code of the function,
                                            including FAIL */
            .
            .
        DETACH(ProgPtr1, Label1)
            .
            .
        DETACH(ProgPtr2, Label2)
            .
            .
            .
        DETACH(ProgPtrN, LabelN)
            .
            .
        /* End of the actual program code */
END

void Cboot(void)
{
        CPRED(Predname, Arity, Funcname, sizeof(State));
}
```

Example

As an example, the count predicate `for_pred(+`*Minimum*`, ?`*Counter*`, +`*Maximum*`)` is to be implemented; its purpose is to unify *Counter* with the value of *Minimum* on the first

call of the function and then unify *Counter* with the next-higher integer each time during backtracking. After *Maximum* has been reached, the predicate should fail.

```
#include "cpred.h"

typedef struct { long curr; long maxim; } FORSTATE;

COROUTINE(Forpred, FORSTATE, state)

RESUME(1, next);

BEGIN
        if (    TermIsInteger(PrologArg(1), &state->curr)
        &&      TermIsInteger(PrologArg(3), &state->maxim))
        {
                while (state->curr <= state->maxim)
                {
                        if (TermUnifyInteger(PrologArg(2), state->curr))
                        {
                                DETACH(1, next);
                        }
                        else PrologUndo();

                        state->curr++;
                }
        }
END

void Cboot(void)
{
        CPRED("for_pred", 3, Forpred, sizeof(FORSTATE));
}
```

Framework for complex C functions with epilog

Following the final exit from the coroutine (in Prolog: `fail`) there may still be certain actions to be performed (e.g. closing files). These actions can be executed in an epilog.

```
#include "cpred.h"

typedef struct { ... } State;

COROUTINE(Funcname,State,StatePtr)
        RESUME(ProgPtr1, Label1);
```

```
        RESUME(ProgPtr2, Label2);
            .
            .
            .
        RESUME(ProgPtrN, LabelN);
        HASEPILOG;

BEGIN
        PrologEnableEpilog();

        /* Start of the actual program code of the function,
                                        including FAIL */

            .
        DETACH(ProgPtr1, Label1)
            .
        DETACH(ProgPtr2, Label2)
            .
            .
            .
        DETACH(ProgPtrN, LabelN)
            .
        /* End of the actual program code */

EPILOG:
        /*  Program code for the epilog */
END

void Cboot(void)
{
        CPRED(Predname, Arity, Funcname, sizeof(State));
}
```

Example

The predicate `readatom(+`*D*`, ?`*Z*`)` is to have the following properties: When called, it opens file *D*, reads the first line from the file, forms a Prolog atom out of it and unifies it with *Z*. During backtracking, it reads the next line each time and passes it to *Z*. At end of file, the next call fails and the file is closed.

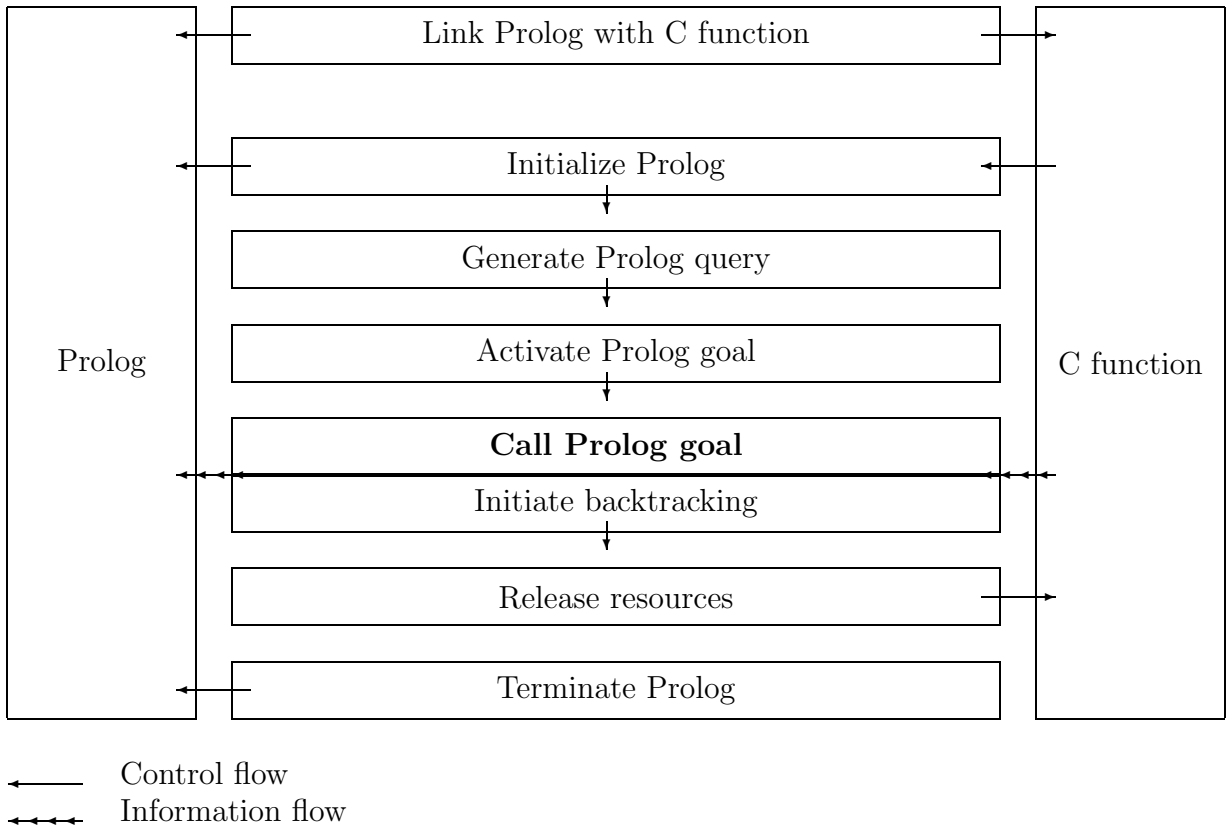
```
#include <stdio.h>
#include "cpred.h"

#define LINELEN 100
```

```
static BOOLEAN readline(FILE *fp, char *line)
{
        int c;
        int i = 0;

        c = getc(fp);

        while ((c != EOF) && (c != '\n') && (i < LINELEN))
        {
                line[i++] = c;
                c = getc(fp);
        }
        line[i] = '\0';
        return c != EOF;
}

typedef struct { FILE *fp; } ReadState;

COROUTINE(Read, ReadState, state)
        RESUME(1, Label1);
        HASEPILOG;
BEGIN
        char line[LINELEN+1];
        STRING name;

        PrologEnableEpilog();

        state->fp = NULL;

        if (!TermIsAtom(PrologArg(1), &name))
        {
                FAIL;
        }
        if ((state->fp = fopen(name, "r")) == NULL)
        {
                ErrorContextSet(EXISTENCE_ERROR,
                        "cannot_open_file", name, 1, NULL);
                FAIL;
        }
        while ( readline(state->fp, line) )
        {
                if ( TermUnifyAtom(PrologArg(2), line))
                        DETACH(1, Label1)
                else PrologUndo();
        }
```

```
EPILOG:
        if (state->fp != NULL)
        {
                fclose(state->fp);
                printf("file closed by epilog\n");
        }
END

void Cboot(void)
{
        CPRED("readatom", 2, Read, sizeof( ReadState ));
}
```

10.2 Passive C interface

The following diagram gives a general overview of the operating principle of the passive C interface.



Functions of the active C interface

You initialize the Prolog system by calling the function `InitProlog()`. At this point you can add to the database predicates which you will need later for executing goals. A Prolog query is initially presented as a C string. You must use the function `ParseProlog()` to convert this string into a goal term, and the function `PrologOpen()` to convert it into an active Prolog goal. The information about an active goal and its processing are kept in a data structure of type `t_cursor`.

During goal execution, the function `PrologFetch()` accesses this information and places the results in the same structures. When processing of the current query is completed, you use the function `PrologClose()` to release the resources associated with it. When you no longer require the passive C interface, you release the memory space occupied by the Prolog system with the function `EndProlog()`.

10.2.1 Overview of the built-in data types and functions

Central data types

Information is exchanged between a C program and the Prolog system using four data structures. The data types on which these are based are defined externally, or in the include file *cpro.h* which resides in the directory *$PROROOT/c*.

- `MODULE` contains information relating to the module whose predicates are accessed by the Prolog query. You cannot access the individual components of the structure.

  ```
  typedef struct _t_module *MODULE;
  ```

- `t_query` is a structure containing a Prolog goal which is ready to execute but inactive. You cannot access the individual components of the structure.

  ```
  typedef struct _t_query *t_query;
  ```

- `t_cursor` is a structure containing an active Prolog goal. You cannot access the individual components of the structure.

  ```
  typedef struct _t_cursor *t_cursor;
  ```

- `t_parse_context` contains information which is placed there after analysis of the C string.

  ```
  typedef struct
  {
          TERM   pc_term;
          TERM   pc_varlist;
          long   pc_position;
          int    pc_errno;
  } t_parse_context;
  ```

You can access the components of the structure `t_parse_context`, where the individual components contain the following after initialization:

Component	Contents
pc_term	The Prolog term which is read
pc_varlist	The list of Prolog variables as a Prolog term in the form `[` *Variablename* `=` *Variable*`, ...]`
pc_position	Position at which reading from the string was terminated
pc_errno	0 or the error number according to `syntax_error/2` if an error has occurred

i If an error occurs during initialization, the components `pc_term` and `pc_varlist` remain undefined.

In addition to the data types described here, you may also use the data types `TERM`, `ARITY` and `STRING` described under the active C interface (section 10.1).

Central functions of the passive C interface

Function	Purpose
InitProlog()	Initialize the Prolog system and define the standard media
ParseProlog()	Parse the goal string and store the information
ParseError()	Determine error description during the analysis
PrologFetch()	Prove a goal
PrologModule()	Determine module
EndProlog()	Release memory space occupied by the Prolog system

Built-in module variables

All built-in predicates are assigned to the `system` or `user` module as a result of the modular organization of the Prolog system. If you want to prove goals within the built-in modules, you can do this by specifying the global variables defined by Prolog system.

Variable	Module
SystemModule	system
UserModule	user

Passive Prolog goals

The passive C interface distinguishes between passive, active and current Prolog goals. A passive Prolog goal is one which, although executable, is not being executed at present; it is, so to speak, on hold. You should create a passive Prolog goal if you subsequently intend to initiate the goal on more than one occasion.

Function	Purpose
QueryCreate()	Create passive Prolog goal
QueryDispose()	Release passive Prolog goal
QueryOpen()	Activate passive Prolog goal

Active Prolog goals

An active Prolog goal is ready for execution. A current goal is also an an active goal, but simply created most recently in the time frame. Only active goals can be proved using the function `PrologFetch()`. You must first make a passive Prolog goal into an active and current goal by means of the function `QueryOpen()`.

Function	Purpose
PrologClose()	Deactivate current Prolog goal
PrologError()	Query error on last call of PrologFetch()
PrologGoal()	Determine term representation of current Prolog goal
PrologOpen()	Create active Prolog goal

Variable instantiations

You can use the following functions to determine the variable instantiation in the current Prolog goal.

Function	Purpose
PrologIsInteger()	Check for integer
PrologIsFloat()	Check for floating-point number
PrologIsString()	Check for string
PrologIsTerm()	Check for term

Functions for formatted output

Function	Purpose
fprintt()	Formatted output of
printt()	Prolog terms and
sprintt()	C data objects
outputlen()	Count the characters for formatted output

10.2.2 Working with the passive C interface

This section starts with a simple framework for the use of the passive C interface. This is followed by a description of the individual steps involved when using it. Reference is made to this framework, and possible extensions to it are shown. Examples are provided to illustrate the description.

Simple framework for the passive C interface

The file *$PROROOT/c/main.c* contains a framework program for using the passive C interface. You can copy this and modify it to suit your own application.

```
#include <stdio.h>
#include <stdlib.h>
#include "cpro.h"

int main(int argc, const char *argv[])
{
    t_parse_context context;        /* If several goals are to be   */
    t_cursor cursor;                /* stored, several variables of */
                                    /* this type must be defined    */

    if ( !InitProlog(argc, argv, stdin, stdout, stderr) )
    {
        if ( PrologInitError != NULL )
            fprintf(stderr, "%s\n", PrologInitError);
            /* Error message on initialization error */
    }
    else
    if (    ParseProlog("program.", &context)
    &&      (cursor = PrologOpen(SystemModule,
                    context.pc_term, context.pc_varlist)) != NULL
    ) {
        (void) PrologFetch(cursor);
        /* The values of the variables in the goal can */
        /* now be used elsewhere in the program        */
        .
        .                       /* Errors detected during call    */
                                /* of the goal can be reported at */
        PrologClose(cursor);    /* this point in the program      */
        EndProlog();            /* Release memory space           */
        return 0;
    }

    fprintf(stderr, "could not initialize Prolog\n");
    return 1;                   /* Program abort on serious */
}                               /* initialization error     */
```

Procedure

- Preparatory steps
  Each source file in which the passive C interface is to be used must contain the following

include statement.

```
#include "cpro.h"
```

- Initializing the Prolog system
  When the Prolog system is initialized, you can specify various options on the operating system command line. These options can be taken from the parameters of the main function by means of `argc` and `argv`. The call

  `InitProlog(`*argc*`, `*argv*`, `*in*`, `*out*`, `*err*`);`

  initializes the Prolog system, the parameters *in*, *out* and *err* define the standard streams for the Prolog system.

- Parsing a Prolog goal
  A goal must exist as a C character string *goalstring*.
  A structure *context* of type `t_parse_context` must be defined in the source text:

  `t_parse_context` *context*`;`

  This structure is passed as the second parameter when the parser is called:

  `ParseProlog(`*goalstring*`, `*context*`);`

  `ParseProlog()` checks whether *context* is a structure of type `t_parse_context` and stores the new information about *goalstring* in the components of this structure (thereby overwriting the old information).
  If a goal belonging to *goalstring* is to be used repeatedly, it need only be parsed once, provided you have not in the meantime used the variable *context* when parsing another goal.

- Creating Prolog goals
  You can now create an active or passive Prolog goal using the components `context.pc_term` and `context.pc_varlist`. The function `QueryCreate()` creates a passive goal which is executable, but should not be called at the moment. You can execute only active goals. You create an active goal by calling the function `PrologOpen()`:

  ```
  cursor = PrologOpen(SystemModule, context.pc_term, context.pc_varlist);
  ```

  You make an active goal from a passive goal by using the function `QueryOpen()`. A current goal is the most recently created active goal.

  The functions `PrologOpen()` and `QueryOpen()` both store information about the active goal in the structure *cursor* of type `t_cursor`. The function `PrologFetch()` accesses this structure.

- Proving a Prolog goal, backtracking
  Calling `PrologFetch(`*cursor*`)` starts or restarts the proof for a goal. The first time that `PrologFetch()` is called, a search is made for the first solution for the goal; each subsequent call searches for the next solution.

  If, after successful proof of a goal, the first solution is to be found again (and if further solutions exist), then the structure `t_cursor` must be recreated by calling `PrologOpen()` or `QueryOpen()` again before calling `PrologFetch()`.

Proof of a goal also restarts with the search for the first solution if, since the last proof, `ParseProlog()` has been called or `PrologFetch()` has been called with a different *cursor*.

If the search for a solution is successful, the return value from `PrologFetch()` is `TRUE`. If no (further) solution can be found, the return value is `FALSE`.

- Checking variable instantiation within a goal
  The value with which a variable of a goal is instantiated after execution is stored as a data object of the type `long`, `double`, `char*` or `TERM` (Prolog term), depending on the selected default. Using the variable name, you can check in the current goal (*cursor*) whether the expected variable is present, and query its value. Depending on the values, you can initiate further actions in the program. In contrast to previous Prolog versions, a type conversion is no longer required.

  ```
  PrologIsTerm(cursor, "term", &term);
  ```

  If you check for a term, you can use the functions `TermType()` or `TermDecompose()` for further analysis of the term:

  ```
  TermDecompose(term, &info);
  ```

- Outputting results
  The three functions `fprintt()`, `printt()` and `sprintt()` are available for formatted output of Prolog terms and C data objects. They correspond to the C functions `fprintf()`, `printf()` and `sprintf()`, respectively.
  The function `outputlen()` is used to determine the number of characters in a term for formatted output.

- Reusing Prolog goals
  As soon as a character string *goalstring* has been parsed successfully, the information about the goal is stored in the structure `t_parse_context`. If several goals are to be stored and reused, your best course of action is to create passive goals. You need to parse a passive goal only once but you can activate it as often as required with the function `QueryOpen()`.

- Terminating Prolog
  When you no longer need the passive C interface, you should release the memory space reserved for the Prolog system by calling `EndProlog()`.

- Compiling and linking
  In order to compile and link the program, you should copy the makefile *model_make* from the directory *$PROROOT/c* into your working directory. Then proceed as described in section 10.4. Set the variable `MAIN` to the absolute pathname of your extended file *main.c* and choose a new name for the extended Prolog system.

  After you have copied the makefile, you can call the program from the shell level using the name defined in the makefile. If you have set the necessary options (see section on initializing the Prolog system above), you can also specify call parameters for the Prolog system.

Example

In the first example the Prolog database is to be used as a data base. The file *supplier.pro* contains the following dispatch list as a Prolog predicate:

```
supplier( grocer, banana, 500, 7.50).
supplier( grocer, apple, 600, 6.50).
supplier( grocer, orange, 20, 5.60).
supplier( grocer, cabbage, 100, 3.45).
supplier( grocer, carrot, 100, 4.50).
supplier( bakery, bread, 3, 7.50).
supplier( bakery, cake, 0, 38.00).
supplier( bakery, cookies, 600, 1.25).
supplier( burger, hamburger, 40, 1.50).
supplier( burger, 'french fries', 40, 1.00).
supplier( burger, 'soft drink', 20, 2.00).
supplier( burger, 'ice cream', 30, 1.25).

:- init((save_system(supplier),halt)).
```

A C program accesses this data and calculates the total price for each article from a given supplier. The supplier is the call argument for the function. The source text for such a function could look something like this:

```
#include <stdio.h>
#include <stdlib.h>
#include "cpro.h"

static BOOLEAN initialized = FALSE;

static void
#if defined(__STDC__)
leave_application(int status)
#else
leave_application(status)
int status;
#endif
{
        if ( initialized )
                EndProlog();
        exit(status);
}

void
```

```
#if defined(__STDC__)
main(int argc, char *argv[])
#else
main(argc, argv)
int argc; char *argv[];
#endif
{
        STRING prolog_args[4];
        char query[100];
        t_parse_context context;
        t_cursor cursor;
        TERMCONTEXT tmcontext;

        if (argc != 2)
        {
                fprintf(stderr, "Usage: %s supplier\n", argv[0]);
                leave_application(1);
        }

        /* The save state supplier.pst is read into the database
           automatically at startup
        */
        prolog_args[0] = argv[0];
        prolog_args[1] = "-r";
        prolog_args[2] = "supplier.pst";
        prolog_args[3] = NULL;

        initialized = InitProlog(3, prolog_args, stdin, stdout, stderr);

        if ( !initialized )
        {
                fprintf(stderr, "PROLOG could not be initialized\n");
                leave_application(1);
        }

        /* argv[1] = supplier */
        sprintf(query, "supplier(%s, Article, Amount, Price), Amount > 0.",
                argv[1]);

        tmcontext = TermContext();
        if ( !ParseProlog(query, &context) )
        {       /* Output information for syntax error */
                fprintf(stderr, "SYNTAX: %s\n", ParseError(context.pc_errno));
                leave_application(1);
        }
```

```
        cursor = PrologOpen(UserModule, context.pc_term, context.pc_varlist);
        if ( cursor != NULL )
        {
                TERM error;

                printf("Delivery from %s:\n\n", argv[1]);
                printf("%-15s %s %9s\n",
                        "Article", "Amount", "Price");
                while ( PrologFetch(cursor) )
                {
                        const char *article;
                        long amount;
                        double price;

                        if (    PrologIsString(cursor,  "Article", &article)
                        &&      PrologIsInteger(cursor, "Amount",  &amount)
                        &&      PrologIsFloat(cursor,  "Price",   &price)
                        ) {
                                printf("%-15s  %3ld  %10.2lf\n",
                                        article, amount, amount*price);
                        }
                }

                if ( PrologError(cursor, &error) )
                        printt("ERROR: %t\n", error);
                PrologClose(cursor);
        }
        /* free terms created by ParseProlog() */
        TermCollect(tmcontext, NULL);

        leave_application(0);
}
```

The program is compiled and linked. It can now be called, for example, as follows:

```
$ supply bakery

Delivery from bakery:

Article  Amount Price

bread     3     7.50
cake      0    38.00
cookies   600   1.25
```

10.3 C device drivers

In IF/Prolog you can write your own device drivers in C and link them into the Prolog system. To do this you require the predefined C functions and C data structures which are contained in the include file *device.h* in the directory *$PROROOT/c*.

10.3.1 t_device structure

You define an input/output device driver in a `t_device` structure which is defined in the file *device.h*:

```
typedef struct
{
        STRING   dev_name;
        BOOLEAN  (*dev_open)(STRING, t_iomode *, t_iounit *);
        BOOLEAN  (*dev_close)(t_iounit, BOOLEAN);
        void     (*dev_stop)(void);

        int      (*dev_read)(t_iounit, void *, int);
        int      (*dev_write)(t_iounit, void *, int);
        BOOLEAN  (*dev_flush)(t_iounit);

        BOOLEAN  (*dev_is_unix_file)(t_iounit, int *);

        const t_control *dev_control;
        const t_control *dev_class_control;
} t_device;
```

The components of the structure determine which facilities the driver is to have. The individual components of the `t_device` structure are described below:

`STRING dev_name;`

The first component of `t_device` is a pointer to a string giving the name of the device driver. You can freely select the name of the driver, but it must be unique, i.e. you must not define any other device driver with the same name. The following device drivers are already built-in in the Prolog system.

`file` Device driver for regular files
`null` Device driver for "bit bucket"
`pipe` Device driver for operating system pipes
`standard` Device driver for standard input, output and error
`string` Device driver for character strings
`socket` Device driver for interprocess communication sockets

`BOOLEAN (*dev_open)(STRING` *Name*`, t_iomode *`*mode*`, t_iounit *`*deviceunit*`);`

`*dev_open()` is a pointer to a function of the driver which opens a stream. If a corresponding stream can be opened, `dev_open()` must return `TRUE`; otherwise, it returns `FALSE`. If no stream is to be opened, you may also specify the `NULL` pointer here.

You call the function later implicitly with the predicates `open/3/4`:

```
open(Device(Name), Mode, Stream)
open(Device(Name), Mode, Stream, Options)
```

The arguments of the predicates `open/3/4` are then mapped to the parameters of the function *`dev_open()`, where *Name* is passed as a string, and the arguments in *Mode* and *Options* are passed as flags.

`STRING` *Name* *Name* is a the name of an input/output stream.

`t_iomode` **mode* This parameter determines the mode in which the stream is to be opened and also the options, if defined (see also the *IF/Prolog Reference Manual* [1], `open/4` predicate). The flags in **mode* can be modified by the device driver. The following flags are available:

`S_IO_READ`
: The stream is opened for reading.

`S_IO_WRITE`
: The stream is opened for writing, the file pointer is set to the start of the file.

`S_IO_APPEND`
: The stream is opened for writing, the file pointer is set to the end of the file.

`S_IO_BINARY`
: The stream is opened for binary data.

`S_IO_REPOSITION`
: You can query the position in the input/output stream and set it at a later time.

`S_IO_NBUF`
: The Prolog system is not to buffer the data.

`S_IO_LBUF`
: The Prolog system is to buffer the data line by line.

`S_IO_FBUF`
: The Prolog system is to buffer all data.

`S_IO_BUFFER`
: This flag is equivalent to logically ORing the buffer flags `S_IO_NBUF`, `S_IO_LBUF` and `S_IO_FBUF` and can be used as a mask.

`S_IO_EA_ERROR`
: If a further read operation is attempted after the end of the file has been reached, an exception is to be generated.

`S_IO_EA_RESET`
: If a further read operation is attempted after the end of the file has been reached, the input/output stream is to be reset.

`S_IO_EA_EOF_CODE`

All predicates which read from the stream behave as though the end of the input has been reached for the first time and will return `end_of_file`.

`S_IO_EOF_ACTION`
This flag is equivalent to logically ORing the input/output flags `S_IO_EA_ERROR`, `S_IO_EA_RESET` and `S_IO_EA_EOF_CODE` and can be used as a mask.

With `open/3` you cannot set any options; consequently, none of the flags `S_IO_NBUF, S_IO_LBUF` and `S_IO_FBUF` is set when *`dev_open()` is called. If your device driver does not set any of these three flags, then the flag `S_IO_FBUF` will be set automatically when *`dev_open()` is exited. Even when there are technical reasons why the device driver itself provides the buffering, you should set **mode* correctly.

If no option is set for `eof_action` in the predicate `open/3/4`, then none of the flags `S_IO_EA_ERROR, S_IO_EA_RESET` and `S_IO_EA_EOF_CODE` is set. However, opening of a stream is rejected when repositioning is requested but is not possible, or when the requested input/output direction cannot be supported.

`t_iounit` **deviceunit*
The pointer *deviceunit* contains the address of the unique identification of the input/output stream. This unique identification is the reference for all subsequent operations on the stream.

`BOOLEAN (*dev_close)(t_iounit` *deviceunit*`, BOOLEAN` *force*`);`
Closes the stream which you have opened with `*dev_open()`. If you do not need `dev_close()`, you may also enter the `NULL` pointer here. If successful, the function must return `TRUE`; otherwise `FALSE`.

`t_iounit` *deviceunit*
Identification of the input/output stream.

`BOOLEAN` *force*
To be set according to the `force` option of the predicate `open/3/4`, but not currently evaluated by the standard device drivers.

`void (*dev_stop)(void);`
Closes the driver. The function is called before exiting from the Prolog system. It enables you, for example, to eliminate any side effects which installation of your driver has caused (e.g. closing all open windows in a windowing system, or releasing memory). You may also enter a `NULL` pointer here.

`int (*dev_read)(t_iounit` *deviceunit*`, void *`*buffer*`, int` *number*`);`
This function reads a maximum of *number* characters from the input/output unit *deviceunit* into a buffer. If successful, *`dev_read()` returns the number of characters read; at the end of the data it returns 0; otherwise -1.

`int (*dev_write)(t_iounit` *deviceunit*`, void *`*buffer*`, int` *number*`);`
This function writes a maximum of *number* characters from a buffer to the input/output unit *deviceunit*. If successful, *`dev_write()` returns the number of characters written; otherwise -1.

`BOOLEAN (*dev_flush)(t_iounit` *deviceunit*`);`
*`dev_flush()` flushes any buffers of the device driver, i.e. any data buffered for the stream is output. In place of this function you may also enter the `NULL` pointer in the

structure `t_device`. If successful, *`dev_flush()` returns `TRUE`; otherwise `FALSE`.

`BOOLEAN (*dev_is_unix_file)(t_iounit` *deviceunit*`, int *`*no*`);`

This function checks whether the input/output unit *deviceunit* is a UNIX file. If so, it returns `TRUE` and writes the UNIX file number of the unit in *no*. If no corresponding file exists, the function returns `FALSE`. If *`dev_is_unix_file()` is always `FALSE`, you may also enter the `NULL` pointer in the structure `t_device` instead.

`const t_control *`*dev_control*`;`

`*dev_control` points to an array of type `t_control`, in which you can place an arbitrary set of stream functions (control functions). These control functions enable you to use built-in predicates to access the device driver which you have programmed yourself. When the predicate `stream_control/2` and certain other predicates are called, their functions are mapped to the functions in *`dev_control`. The predicates for which you may define control functions are given in the table in section 10.3.2.

`t_control` has the following components:

```
typedef struct
{
        STRING  ctl_name;
        ARITY   ctl_arity;
        BOOLEAN (*ctl_handler)(t_stream, t_iounit, TERM *);
} t_control;
```

`STRING` *ctl_name*

Functor of the control information as a string. You mark the end of the array containing control functions with a `NULL` pointer in this component.

`ARITY` *ctl_arity*

Arity of the control information.

`BOOLEAN (*ctl_handler)(t_stream` *stream*`, t_iounit` *deviceunit*`, TERM *`*term*`);`

Control function which is to perform a particular function on a stream or on an input/output unit. It takes as its arguments a stream and its associated input/output unit. The control information is passed by Prolog as an array of terms. You can access the terms using the macro `CTLARG(n)`. If successful, the function *`ctl_handler()` returns the value `TRUE`; otherwise `FALSE`.

If you do not wish to provide any control functions, you may also enter the `NULL` pointer in the structure `t_device`.

`const t_control *`*dev_class_control*`;`

`*dev_class_control` points to an array of type `t_control`, in which you can place an arbitrary set of class-specific control functions. These class-specific control functions, which are not dependent on any concrete stream, enable you to use the built-in predicates `device_control/2` and `current_device_control/3` to access the device driver which you have programmed yourself. When the predicates are called, their functions are mapped to the functions in *`dev_class_control`, where the component *stream* is always initialized with the `NULL` pointer.

You can use the same control function for handling both class-specific and stream-specific functions. Through the component *stream*, the function can check the purpose

for which it has been activated.

10.3.2 Control functions

A device driver provides a set of commands for performing special operations on input/output streams. If you define the required control commands, the built-in predicates which use the control interface of device drivers can also support the input/output streams for a device driver which you have programmed yourself.

The built-in device drivers implement the following control commands:

The following control commands are implemented in the built-in device drivers of the Prolog system:

Device	Command	Meaning
`file`	`stream_type(`*Arg*`)`	see `stream_type/2`
	`stream_size(`*N*`)`	Unifies *N* with the size of the file
	`isatty`	`stream_control/2` succeeds if *Stream* is a terminal
	`fileno(`*N*`)`	Unifies *N* with the operating system dependent file number of the file
	`clear_eos`	see `clear_eos/0/1`
	`clear_eol`	see `clear_eol/0/1`
	`clear_screen`	see `clear_screen/0/1`
	`tty_size(`*Z*`,`*S*`)`	see `tty_size/2/3`
	`move_cursor(`*Z*`,`*S*`)`	see `move_cursor/2/3`
	`seek(`*P*`)`	see `set_stream_position/2`
	`tell(`*P*`)`	see `stream_property/2`
`standard`	`stream_type(`*Arg*`)`	see `stream_type/2`
	`stream_size(`*N*`)`	Unifies *N* with the size of the file
	`isatty`	`stream_control/2` succeeds if *Stream* is a terminal
	`fileno(`*N*`)`	Unifies *N* with the operating system dependent file number of the file
	`clear_eos`	see `clear_eos/0/1`
	`clear_eol`	see `clear_eol/0/1`
	`clear_screen`	see `clear_screen/0/1`
	`tty_size(`*Z*`,`*S*`)`	see `tty_size/2/3`
	`move_cursor(`*Z*`,`*S*`)`	see `move_cursor/2/3`
`pipe`	`stream_type(`*Arg*`)`	see `stream_type/2`
	`fileno(`*N*`)`	Unifies *N* with the operating system dependent file number of the file
`socket`	`stream_type(`*Arg*`)`	see `stream_type/2`
	`fileno(`*N*`)`	Unifies *N* with the operating system dependent file number of the file

`string`	`stream_type(`*Arg*`)`	see `stream_type/2`
	`stream_size(`*N*`)`	Unifies *N* with the size of the string
	`string(`*Arg*`)`	Unifies *Arg* with the contents of the string
	`seek(`*P*`)`	see `set_stream_position/2`
	`tell(`*P*`)`	see `stream_property/2`
`null`	`stream_type(`*Arg*`)`	see `stream_type/2`
`-`	`get_last(`*Z*`,`*S*`)`	see `get_last/2/3`

There are a number of predicates which are mapped to special control functions. If you wish to support these predicates with the device driver, you must specify the corresponding control functions for the input/output unit.

Predicate	**Control function**	**Arity**
clear_eol/0/1	clear_eol	0
clear_eos/0/1	clear_eos	0
clear_screen0/1	clear_screen	0
stream_copy/2	dup	0
stream_copy/2	fileno	1
get_last/2/3	get_last	2
move_cursor/2/3	move_cursor	2
set_stream_position/2	seek	1
stream_type/2	stream_type	1
stream_property/2	tell	1
tty_size/2/3	tty_size	2

The following predicates allow you to access device drivers:

Predicate	**Meaning**
current_device_control(+*Device*, ?*Command*, ?*Arity*)	Query information about device drivers
current_stream_control(@*Stream*, ?*Command*, ?*Arity*)	Query information on device drivers
device_control(+*Device*, @*Command*)	Control an input/output device
stream_control(@*Stream*, @*Command*)	Control an input/output stream
stream_device(@*Stream*, ?*Device*)	Determine device name of an input/output stream
stream_property(?*Stream*, ?*Info*)	Query information on input/output streams
stream_type(@*Stream*, ?*StreamType*)	Query type of an input/output stream

10.3.3 Functions for implementing device drivers

The C interface supports the implementation of device drivers with the following functions.

Linking a device driver into Prolog

The function `DeviceCreate()` declares the new device driver to the Prolog system. The function has as its argument a pointer to a structure of type `t_device` in which you have defined all the properties of the new device driver.

```
#include "device.h"
extern void DeviceCreate(const t_device *device);
```

`const t_device` **device* is a pointer to the structure of type `t_device` which contains all the properties of the new device driver. You should create this structure with the property `static const` and name it. In this structure you must specify everything that the new device driver is to do.

You link the new device driver into a new Prolog system by means of the function `Cboot()`. See page 238 for information on creating a new Prolog system.

Example

```
#include "device.h"

static const t_device file_device =
{
        ...
};

void Cboot(void)
{
        DeviceCreate(&file_device);
        ...
}
```

Flushing Prolog buffers

The function `StreamFlush()` flushes the buffers provided for the C interface by the Prolog system. If the buffers have been flushed, the function returns the value `TRUE`; otherwise `FALSE`.

```
#include "device.h"
extern BOOLEAN StreamFlush(t_stream stream, long *number);
```

`t_stream` *stream*	is a pointer to the `stream` which is passed to a control function (see *`ctl_handler()`) as the first argument.
`long` **number*	This variable contains the *number* of characters which were in the buffer prior to flushing.

Querying stream mode

The function `StreamMode()` returns as a flag array the mode of a stream (see also `open/3/4`).

```
#include "device.h"
extern t_iomode StreamMode(t_stream stream);
```

`t_stream` *stream*	is the stream whose mode is to be queried.

Example

This example implements a simple device driver which can support UNIX files and which, in addition to the functions for opening, closing, reading and writing, contains two control functions.

```
#include <string.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/stat.h>
#include "device.h"

static int
file_read(t_iounit unit, void *buffer, int nbytes)
{
        return fread(buffer, 1, nbytes, (FILE*)unit);
}

static int
```

```
file_write(t_iounit unit, void *buffer, int nbytes)
{
        return fwrite(buffer, 1, nbytes, (FILE*)unit);
}

static BOOLEAN
file_flush(t_iounit unit)
{
        fflush((FILE*)unit);
        return TRUE;
}

static BOOLEAN
file_close(t_iounit unit, BOOLEAN force)
{
        fclose((FILE*)unit);
        return TRUE;
}

static BOOLEAN
file_is_unix(t_iounit unit, int *no)
{
        *no = fileno((FILE*)unit);
        return TRUE;
}

static BOOLEAN
file_open(const char *filename, t_iomode *iomode, t_iounit *unit)
{
        FILE *fd;
        char mode[3];

        mode[1] = mode[2] = '\0';

        switch (*iomode & (S_IO_READ|S_IO_WRITE|S_IO_APPEND))
        {
        case S_IO_READ:
             mode[0] = 'r';
             break;
        case S_IO_WRITE:
             mode[0] = 'w';
             break;
        case S_IO_WRITE|S_IO_APPEND:
             mode[0] = 'a';
             break;
```

```
        default:
             return FALSE;
        }

        if ((*iomode & S_IO_BINARY) != 0) mode[1] = 'b';

        if ((fd = fopen(filename, mode)) != NULL)
        {
             *unit = (t_iounit)fd;
             return TRUE;
        }
        return FALSE;
}

CONTROL(file_isatty,stream,unit)
{
        return isatty(fileno((FILE*)unit));
}

CONTROL(file_size,stream,unit)
{
        struct stat finfo;

        if ( fstat(fileno((FILE*)unit), &finfo) == -1)???
                return FALSE;
        if ((finfo.st_mode & S_IFMT) == S_IFREG)
                return TermUnify(CTLARG(1),(long)finfo.st_size);
        return FALSE;
}

static const t_control l_file_control[] =
{
        { "stream_size",  1, file_size },
        { "isatty",       0, file_isatty },
        { NULL }
};

static const t_device l_file_device =
{
        "my_device",
        file_open,
        file_close,
        NULL,
        file_read,
        file_write,
```

```
        file_flush,
        file_is_unix,
        l_file_control
        NULL
};

void Cboot(void)
{
        DeviceCreate(&l_file_device);
}
```

10.4 Linking Prolog and C

If you wish to use one of the C interfaces and have performed the steps described in the previous sections, you must generate a new extended Prolog system which will then contain the C functions.

- To do this you must first write a `Cboot()` function containing the following calls.
  - a call to the function `CPRIM()` for each simple function which is to be linked or for a Prolog call in a C program;
  - a call to the function `CPRED()` for each complex C function;
  - a call to the function `MCPRIM()` for each simple function which is to be assigned to a particular Prolog module;
  - a call to the function `MCPRED()` for each complex C function which is to be assigned to a particular Prolog module; and
  - a call to the function `DeviceCreate()` for each C device driver which is to be connected to Prolog.

  You thus have the following functions at your disposal with which you can link C functions in Prolog and Prolog calls in C programs

C function	**Purpose**
CPRIM()	Add simple C function in Prolog
CPRED()	Add complex C function in Prolog
MCPRIM()	Assign simple C function to a Prolog module
MCPRED()	Assign complex C function to a Prolog module
DeviceCreate()	Link C device driver into Prolog

i You may define only one `Cboot()` function.

- You should then copy the model makefile *model_make* from the directory *$PROROOT/c*, and modify it to suit your application. With this makefile you can create a new Prolog application.

  The libraries and objects which you select, and the order in which you specify them, are significant and can vary according to the system.

Example

In the following you will see the contents of the makefile *model_make* for a UNIX system. The contents of the makefile may differ for different machines.

```
# This Makefile is a model for making your own application kernel,
# using the C-Language-Interface of PROLOG
# and possibly using Motif or database interfaces or Constraint Package
#
# Change the definitions if appropriate
# COMMANDS

# The PROROOT is the directory where your PROLOG is installed
PROROOT=DEFAULTROOT

# These library definitions are always necessary
PROLIB_ASCII=$(PROROOT)/c/libAprolog.a
PROLIB_MOCTET=$(PROROOT)/c/libKprolog.a
PROLIB_CHARS=$(PROLIB_MOCTET)   # Multibyte character support
PROLIB_KERNEL=$(PROROOT)/c/libprolog.a
PROLIB_DUMMY=$(PROROOT)/c/libdummy.a

# Comment the following library inclusion if you explicitly wish to
# exclude multiple precision integer and rational number support
# The library must be included if you include the constraint libraries
PROLIB_RAT=$(PROROOT)/c/libRprolog.a

# These library definitions are only necessary, if the application
# uses the Motif Interface of PROLOG
# Uncomment the appropriate lines!
#
# Motif 1.1
#PROLIB_MOT=$(PROROOT)/c/promotif11.o
#PROLIB_MOT=$(PROROOT)/c/libMot11prolog.a
#SYSLIBS_MOT=-lMrm -lXmu -lXm -lXt -lX11
# Motif 1.2
#PROLIB_MOT=$(PROROOT)/c/promotif12.o
#PROLIB_MOT=$(PROROOT)/c/libMot12prolog.a
#SYSLIBS_MOT=-lMrm -lXmu -lXm -lXt -lX11
```

```
# These library definitions are only necessary, if the application
# uses the Informix Interface of PROLOG
# Uncomment the appropriate lines!
#
#INFORMIXDIR=/opt/lib/informix
# Informix 4.1
#PROLIB_INF=$(PROROOT)/c/libInf41prolog.a
#SYSLIBS_INF=-L $INFORMIXDIR/lib -lsql4
# Informix 5.0
#PROLIB_INF=$(PROROOT)/c/libInf50prolog.a
#SYSLIBS_INF=-L $INFORMIXDIR/lib/esql -lsql -lgen -los
# Informix 7.1
#PROLIB_INF=$(PROROOT)/c/libInf71prolog.a
#SYSLIBS_INF=-L $INFORMIXDIR/lib/esql -lsql $INFORMIXDIR/lib/netlib.a \
#       -lgen -los

# These library definitions are only necessary, if the application
# uses the Constraints Package of PROLOG
# Uncomment the following lines if necessary
#
#PROLIB_CONS=$(PROROOT)/c/libCprolog.a $(PROROOT)/c/libCDprolog.a \
        $(PROROOT)/c/libCFprolog.a $(PROROOT)/c/libCLprolog.a

PROLIBS_CONFIG=$(PROLIB_MOT) $(PROLIB_INF) $(PROLIB_CONS) $(PROLIB_RAT)
SYSLIBS_CONFIG=$(SYSLIBS_MOT) $(SYSLIBS_INF)

SYSLIBS=

# This defines the objects and/or libraries of your application
USEROBJS=
USERLIBS=

# This defines the PROLOG-Part of your application
PROSOURCE=
SAVESTATE=

# This is the name of your application kernel
TARGET=application

# This is the main module of the application.
# Change this definition for your own main module
MAIN=$(PROROOT)/c/main.o

.c.o:
```

```
        $(IF_cc) -c $(CFLAGS) $<

# This makefile entry generates the application kernel
# according to the above definitions

$(TARGET):      $(MAIN) $(USEROBJS) $(USERLIBS) $(SAVESTATE)
        $(IF_cc) $(LDFLAGS) $(MAIN) \
                $(USEROBJS) $(USERLIBS) \
                $(PROLIB_KERNEL) $(PROLIB_CHARS) \
                $(PROLIBS_CONFIG) \
                $(PROLIB_DUMMY) \
                $(SYSLIBS_CONFIG) $(SYSLIBS) -o $@

# This makefile entry may be used to generate the PROLOG-Part
# of the application
#
#$(SAVESTATE):  $(PROSOURCE)
#       $(PROROOT)/bin/prolog -root $(PROROOT) -r runtime.pst \
#               -st - -c $(PROSOURCE)

clear:
        rm -f $(TARGET) $(USEROBJS) $(SAVESTATE)
```

A new Prolog system (*application*) has been defined in this makefile.

- With the following system call from the shell level you generate a new Prolog system:

  $ make -f model_make ↩

- You can then call the extended Prolog system from the shell level as follows, using the name defined in the makefile (*application*):

  $ application ↩

If you have set the necessary options, you can also specify parameters for the Prolog system in the call (see section 4.1.1).

On starting, the Prolog system calls the `Cboot()` function specified by you and connects the C functions as predicates to Prolog. A newly incorporated predicate now behaves like a built-in predicate of the corresponding type. Should you wish to connect further C functions at some later time, you must extend the function `Cboot()` and your makefile appropriately and generate a new extended Prolog system.

Chapter 11

Graphical User Interface for MS–Windows

11.1 The IF/Prolog Graphical User Interface

IF/Prolog provides a graphical user interface under MS–Windows '95, '98, NT and future releases of MS–Windows operating systems. The following section describes the user interface features and how to use it for developing IF/Prolog applications. After starting the IF/Prolog application the main window appears on the screen. It contains:

- **Menu Bar**
- **Toolbars**
- **Prolog Console with Scrollbars**
- **Status Bar**

Other topics on User Interface features and Application Development includes:

- **Database Browser**
- **Options Dialog**
- **Debugger Support**
- **Linking Custom IF/Prolog Applications with GUI**

This is the IF/Prolog Main Window:

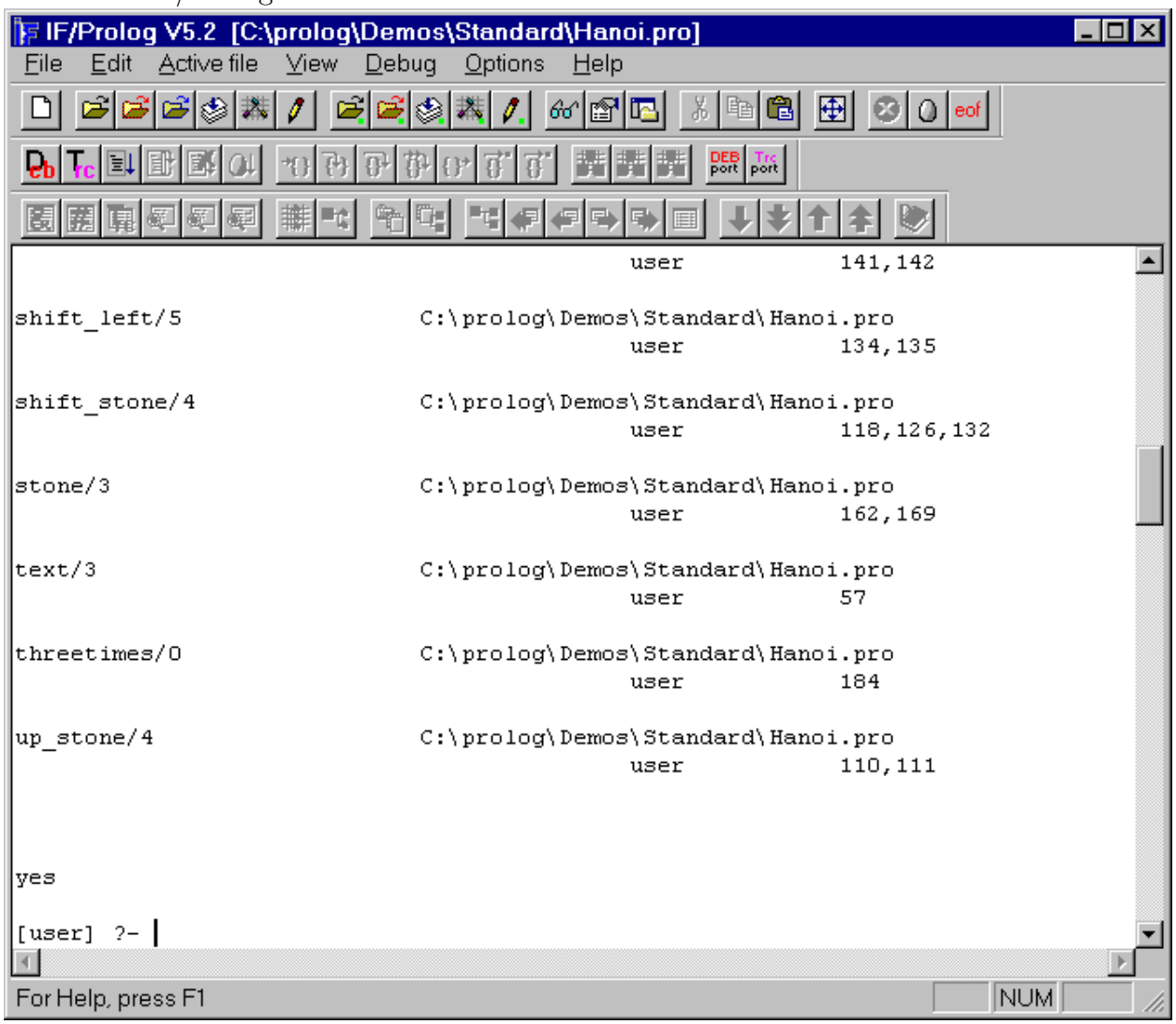


Figure: IF/Prolog Main Window

11.2 IF/Prolog Main Window Menu Bar

This is the IF/Prolog Main Window Menu Bar:

Figure: IF/Prolog Main Window Menu Bar

The Main window menu bar contains several menus:

- **File Menu** File commands - open, consult, reconsult, load, ...
- **Edit Menu** Clipboard commands - cut, copy & paste.
- **Active File Menu** File commands - open, consult, reconsult, load, ... but only for IF/Prolog active file.
- **View Menu** View commands - enable/disable toolbars, database browser or status bar.
- **Debug Menu** Debugger commands - step, skip, continue, ...
- **Options Menu** Options commands - general options, prolog flags, ...
- **Help Menu** Help commands - GUI reference, Prolog reference, ...

11.2.1 File Menu

- **Edit new** Opens a file named "untitled.pro" in the IF/Prolog default editor.
- **Edit** Opens a file dialog. When the file is selected it is open in the IF/Prolog default editor.
- **Consult** Opens a file dialog. When the file is selected it is consulted into the Prolog database.
- **Reconsult** Opens a file dialog. When the file is selected it is re-consulted into the Prolog database.
- **Load** Opens a file dialog. When the file is selected it is loaded into the Prolog database.
- **Compile** Opens a file dialog. When the file is selected it is compiled to the filename with extension *.CMP.
- **Cross Reference** Opens a file dialog. When the file is selected Cross Reference tool is started on this file. The Cross Reference tool output is displayed in the console window.
- **Working directory** Opens a directory dialog. Sets the current working directory.
- **Set Active file** Opens a file dialog. When the file is selected it is set as the IF/Prolog active file (see Active File Menu).
- **Exit** Exits IF/Prolog application.

11.2.2 Edit Menu

- **Cut** Cuts the current selection in the Console window and copies it into the Clipboard.

Note: Selection is cut only if it is a part of an editable section in the console window (e.g. command line, ...). Selection is always copied into the Clipboard.

- **Copy** Copies the current selection in the Console window into the Clipboard.
- **Paste** Pastes the Clipboard (text only) to the Console window.

Note: Selection is pasted only if it is a part of an editable section in the console window (e.g. command line, ...).

11.2.3 Active File Menu

- **Set Active file** Opens a file dialog. When the file is selected it is set as the IF/Prolog active file.
- **Edit** Opens the IF/Prolog active file in the default editor.
- **Consult** Consults the IF/Prolog active file into the Prolog database.
- **Reconsult** Re-consults the IF/Prolog active file into the Prolog database.
- **Compile** Compiles the IF/Prolog active file into the filename with the extension *.CMP.
- **Cross Reference** Starts Cross Reference tool on the IF/Prolog active file.

11.2.4 View Menu

- **Main Toolbar** Activates or deactivates IF/Prolog main toolbar
- **Debug Main Toolbar** Activates or deactivates IF/Prolog main debugger toolbar
- **Debug Information Toolbar** Activates or deactivates IF/Prolog second debugger toolbar
- **Status Bar** Activates or deactivates IF/Prolog status bar
- **IF/Prolog Database Browser** Activates or deactivates IF/Prolog database browser

11.2.5 Debug Menu

- **Abort** Terminates interactive debugger.
- **Continue** Executes debugger to next breakpoint.
- **Step** Processes to next monitored port.
- **Skip** Skips monitoring of a current subgoal.
- **Fast Skip** Skips fast (fast-skip) monitoring of current subgoal.
- **Nonstop** Executes to the end of proof.
- **Force to Succeed** Forces success of current subgoal.
- **Force to Fail** Forces failure of current subgoal.
- **Debug Ports** Turns port monitoring on/off (in the debug).
- **Trace Ports** Turns port monitoring on/off (in the trace).
- **View is a pop-up submenu with the items:**
- **Goal Box** Displays goal box (AND box) in output window.
- **Clause Box** Displays clause box (OR box) in output window.
- **Constraints** Displays all constraints in output window.
- **Breakpoints (Stops)** Displays list of all explicit breakpoints in output window.
- **Trace** Displays trace field in output window.

See Also
Debugger commands, Debugger Toolbar, Debugger Information Toolbar, Console Window, Debugger Support

11.2.6 Options Menu

IF/Prolog Options dialog window allows to set various parameters. It has several subdialogs - property pages:

- **General** Opens the General property page in IF/Prolog Options dialog.
- **Prolog Flags** Opens the Prolog Flags property page in IF/Prolog Options dialog.
- **Preprocessor Options** Opens the Preprocessor Options property page in IF/Prolog Options dialog.
- **Configuration** Opens the Configuration property page in IF/Prolog Options dialog.
- **Cross-Reference Options** Opens the Cross-Reference Options property page in IF/Prolog Options dialog.

11.2.7 Help Menu

- **User Interface** Displays the information on user interface.
- **IF/Prolog Database Browser** Displays the information about IF/Prolog Database Browser.
- **Debugger** Displays the information on debugger user interface.
- **Prolog Reference** Displays on-line predicate reference.
- **About** Displays a short information about the IF/Prolog application.

11.3 Main Window Toolbars

These toolbars are the IF/Prolog Main Window Toolbars:

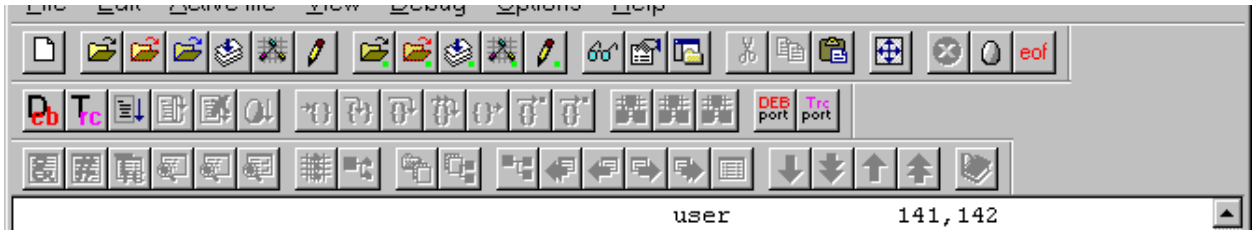


Figure: IF/Prolog Main Window Toolbars

Toolbar buttons are "shortcuts" for most frequent commands. You can invoke various IF/Prolog actions just by button click. You do not need to remember each button functionality. If you move mouse cursor over any toolbar button a short button help text appears over the button. Also the main window status bar displays additional information on button command.

There are three toolbars contained in the IF/Prolog main window:

- **Main Toolbar**
- **Debug Main Toolbar**
- **Debug Information Toolbar**

You can hide or show each of these toolbars by menu commands from the View Menu.

11.3.1 Main Toolbar

This is the Main IF/Prolog toolbar:

Figure: Main IF/Prolog toolbar

Main IF/Prolog toolbar contains buttons for general commands such as file handling (compile, consult, ...), clipboard functions (cut, copy and paste), execution handling (interrupt, break, ...) and tools (database browser, options).

This is a list of toolbar buttons with descriptions:

Button	Command	Description
	Create New	Create new file end open it in the default editor (file is named untitled.pro)
	Consult File	Consult a file and set it as the IF/Prolog active file
	Reconsult File	Reconsult a file and set it as the IF/Prolog active file
	Load File	Load a compiled file
	Compile File	Compile a file and set it as the IF/Prolog active file
	Cross Reference tool on File	Run Cross Reference Tool on a file
	Edit File	Open a file in the default editor and set it as the IF/Prolog active file
	Consult Active File	Consult IF/Prolog active file
	Reconsult Active File	Reconsult IF/Prolog active file
	Compile Active File	Compile IF/Prolog active file
	Cross Reference tool on Active File	Run Cross Reference Tool on IF/Prolog active file
	Edit Active File	Open IF/Prolog active file in default editor
	View IF/Prolog Database	Activate IF/Prolog Database Browser

Button	Command	Description
	General Options	Activate IF/Prolog Options dialog with General property page
	Working Directory	Set working directory
	Cut	Cut current selection and copy it to the Clipboard
	Copy	Copy current selection to the Clipboard
	Paste	Paste text from the Clipboard
	Fit Console Size	Resize main window to fit console size parameters
	Interrupt	Interrupt goal execution
	Break	Break IF/Prolog input loop
eof	End Of File	End IF/Prolog input loop

11.3.2 Debug Main Toolbar

This is the Main IF/Prolog debugger toolbar:

Figure: Main IF/Prolog debugger toolbar

Main IF/Prolog debugger toolbar contains buttons for debugger commands such as debugger/tracer activation, debugger control (abort, step, skip, continue, ...) , branching (back_parent, ...) and debug/trace options.

This is a list of toolbar buttons with descriptions:

Button	Command	Description
	Debug on/off	Activate/deactivate debug mode
	Trace on/off	Activate/deactivate trace mode
	Debug goal	Activate debugger with a goal and set goal as an active debugger goal
	Debug last goal	Activate debugger with an active debugger goal

Button	Command	Description
	Abort	Terminate interactive debugger
	Break inside debugger	Call new input loop
	Continue	Continue to next breakpoint
	Step	Execute up to next monitored port
	Skip	Execute up to exit of a subsequent subgoal
	Fast skip	Accelerate execution to exit of the subsequent subgoal
	Nonstop	Continue without stopping
	Force to succeed	Force current subgoal to succeed
	Force to fail	Force current subgoal to fail
	Back clause	Branch back to clause head
	Back parent	Branch back to CALL port of parent goal
	Back subgoal	Branch back to CALL port of subgoal
DEB port	Set debugger ports	Configure the debugger
Trc port	Set tracer ports	Configure the tracer

See Also: Debugger Commands

11.3.3 Debug Information Toolbar

The IF/Prolog debugger toolbar with additional information on debugging:

Figure: IF/Prolog debugger toolbar with additional information on debugging

Second IF/Prolog debugger toolbar contains buttons for debugger commands that provides additional information on debugging such as different views (goal, clause, ...), structure lists (breakpoints, modules, constraints, ...), screen history, debugger help.

This is a list of toolbar buttons with descriptions:

Button	Command	Description
	Display the goal box	Display the goal box in the window
	Display the clause box	Display the clause box in the window

Button	Command	Description
	Trace current port	Logging passes through ports
	View trace	Display the trace field in the window
	Display explicit breakpoints	Display all explicit brakpoints
	Breakpoints dialog	Invoke breakpoints dialog - it displays the list of breakpoints and allows removing, activating, deactivating or setting the breakpoints
	Display constraints	Displays a pending constraint subgoal in the context of ist definition
	View variable constraints	Displays all suspended constraints at the current breakpoint. Variables that do not occur in the current clause are displayed using their internal name.
	View module	Displays the predicates of the current module from the Prolog database.
	View predicate	Displays the current predicate from the Prolog database.
	View ancestors	Enables to user to show the current dynamic call chain.
	View first screen	Displays the first goal or clause box stored.
	View previous screen	Displays the previous goal or clause box stored.
	View next screen	Displays the next goal or clause box stored.
	View last screen	Displays the last goal or clause box stored.
	View history	Displays the goal or clause box in the history list.
	Line forward	Scrolls line by line forward.
	Page forward	Scrolls page by page forward.
	Line back	Scrolls line by line backward.
	Page back	Scrolls page by page backward.
	Help	Displays the on-line help page on debugger commands and debugger support.

See Also: Debugger Commands

11.4 Prolog Console with Scrollbars

This is the IF/Prolog Main Window Console with Scrollbars:

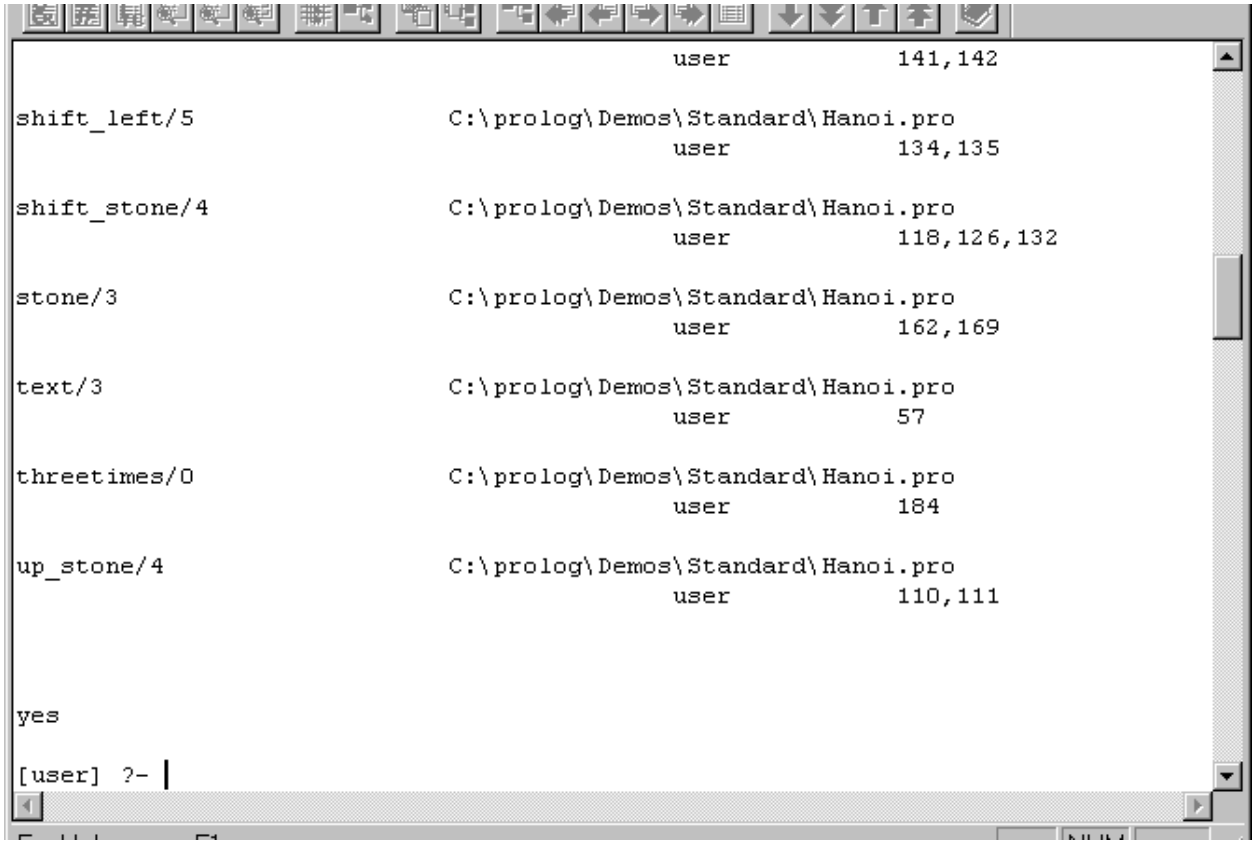


Figure: IF/Prolog Main Window Console with Scrollbars

Console Window is standard input/output device for IF/Prolog. It allows you also to exchange text between the Clipboard and Console window. If you click on the right mouse button the pop-up edit menu appears:

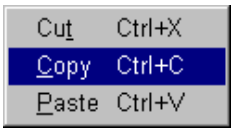


Figure: Console Window Pop-Up menu

You can use left or bottom scrollbars to move displayed text and to show previous output. Console window has also several key sequences that can be used to launch IF/Prolog commands.

11.4.1 Console Key Sequences

Key Sequence	Description	Note
Ctrl + D	EOF - end_of_file/0	development GUI only
Ctrl + N	New file	development GUI only
Ctrl + O	Open file in default editor	development GUI only
Ctrl + Break	Interrupt goal execution	development GUI only
F10	Debugger step	debugger command, development GUI only
Shift + F10	Debugger skip	debugger command, development GUI only
Shift + F11	Debugger fast skip	debugger command, development GUI only
F5	Debugger continue	debugger command, development GUI only
Shift + F5	Debugger abort	debugger command, development GUI only
Ctrl + Insert	Edit copy	
Ctrl + C	Edit copy	
Shift + Ins	Edit paste	
Ctrl + V	Edit paste	
Shift + Delete	Edit cut	
Ctrl + X	Edit cut	
Ctrl + Up	Move back in the command history	
Ctrl + Down	Move forward in the command history	
Home	Go to the beginning of the current input line	
Ctrl + Home	Go to the beginning of the current input	
End	Go to the end of the current input line	
Ctrl + End	Go to the end of the current input	
Left	Go left in the current input	
Right	Go right in the current input	
Up	Go up in the current input	
Down	Go down in the current input	
Esc	Erase current input	

Key Sequence	Description	Note
Del	Delete character	
Backspace	Delete left character	
F1	Help	development GUI only

11.5 Status Bar

This is the IF/Prolog Main Window Status Bar:

Figure: IF/Prolog Main Window Status Bar

Status Bar displays contains a text area and keyboard indicators. When you move the mouse cursor over a toolbar button or select a menu item Status Bar displays short help information to corresponding button or menu command. Keyboard indicators shows the current keyboard mode - Num Lock, Caps Lock or Scroll Lock.

11.6 Database Browser

IF/Prolog Database Browser helps to maintain large Prolog applications. It can show each loaded or consulted module, its predicates, predicates visible in this module, module variables and operators. When available Database Browser displays listing for the module or particular predicates and operators (only available when module or predicate is consulted in debug mode). For system predicates the manual page is shown.

This is the IF/Prolog Database Browser Window:

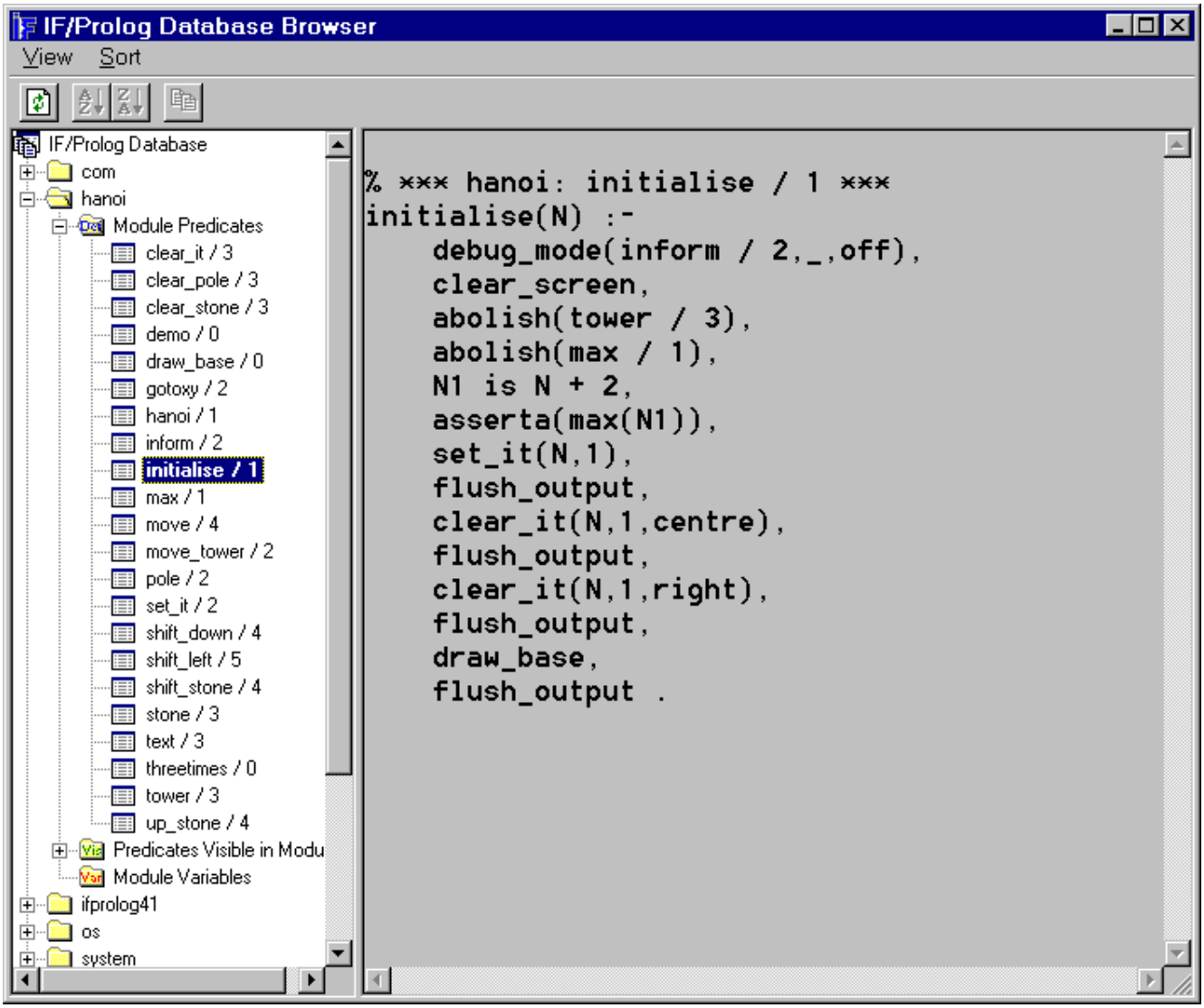


Figure: IF/Prolog Database Browser Window

11.6.1 Database Browser Window

Database Browser window contains several items:

- **Database Tree Pane** Left subwindow of the Database Browser window.
- **Database Item Pane** Right subwindow of the Database Browser window.
- **Menu** Menu at the window top.
- **Toolbar** Toolbar at the window top.

11.6.2 Database Tree Pane

The root item of the tree is the database itself. Database Tree Pane shows all the modules sorted alphabetically as subtrees of the root. One additional module is created for operators. It is situated at the tree bottom. Each module subtree contains three subtrees: Predicates contained in the module, Predicates visible in the module and Module variables.

11.6.3 Database Item Pane

Database Item Pane displays the information for every item that is selected in the left pane - Database Tree Pane.

The Table shows a correspondence between a selected item in the Database Tree Pane and information displayed in the Database Item Pane.

Item (Database Tree Pane)	**Item information (Database Item Pane)**
Item (Database Tree Pane)	Item information (Database Item Pane)
Database (root)	IF/Prolog statistics
Module	Module listing
Module Predicates	List of module predicates
Visible Predicates	List of predicates visible in the module
Module Variables	List of module variables
Predicate	Listing of the predicate or its manual page
Variable	Variable description and value
Operators	List of all operators
Operator	Operator description (type, precedence, ...)

11.6.4 Menu

Database Browser Menu contains several commands useful when using Database Browser:

- **View is a pop-up sub menu contains items:**
- **Refresh** Updates the database browser directories and displays the root item.
- **Exit** Exits the database browser, main IF/Prolog window is activated.
- **Sort is a pop-up sub menu contains items:**
- **Sort ascending** When the right pane item is list (predicates, variables, operators) you can sort it in ascending order.
- **Sort descending** When the right pane item is list (predicates, variables, operators) you can sort it in descending order.
- **Help menu item displays this help page on database browser.**

Toolbar Database Browser Menu contains several buttons useful when using Database Browser:

- **Refresh** Updates the database browser directories and displays the root item.
- **Sort ascending** When the right pane item is list (predicates, variables, operators) you can sort it in ascending order.
- **Sort descending** When the right pane item is list (predicates, variables, operators) you can sort it in descending order.
- **Copy** When the right pane item is text you can select a part of it and copy it to the clipboard.

11.7 Options Dialog

Options Dialog is a tool that allows to modify IF/Prolog configuration or set properties of IF/Prolog system. It includes several panes - property pages:

- **General** General property page includes general settings - default editor, console font and console sizes, startup parameters and current configuration of the startup parameters.
- **Prolog Flags** Prolog flags configuration.
- **Preprocessor Options** Prolog preprocessor configuration. Prolog preprocessor can be configured to preprocess files before consulting or compiling.
- **Configuration** IF/Prolog configuration. Modify stack sizes, memory parameters, ...
- **Cross-Reference Options** Cross-Reference tool configuration.

11.7.1 General Options

This is the General property page:

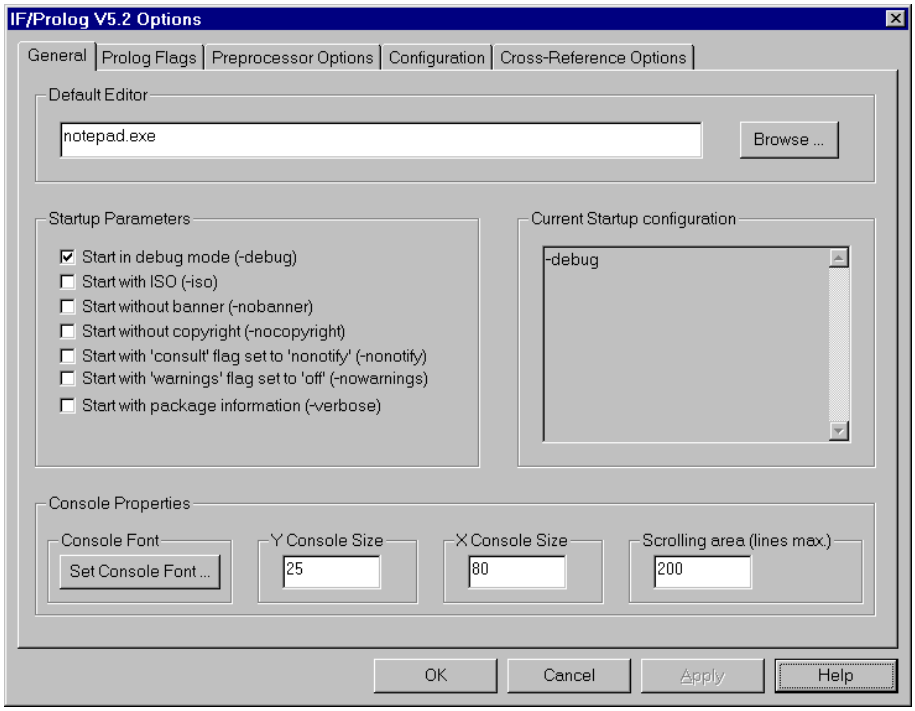


Figure: General property page

General property page includes general settings - default editor, console font and console sizes, startup parameters and current configuration of the startup parameters.

Option	Description
Default editor	Default editor is used by predicate edit/0/1 or by menu (button) commands that edit a file. It can be set by typing a new filename or by selecting a new file in file dialog (available on "Browse..." button click).
Start in debug mode	Starts IF/Prolog with debug flag set to on. This means that all predicates which you read into the database with consult/1, reconsult/1 or assert/1/2 are implicitly declared as dynamic, thereby supporting the debugging of a program.
Start with ISO	Starts IF/Prolog with ISO flag. Only the facilities required to ISO-Prolog are available.
Start without banner	Starts IF/Prolog without a banner window showing a license information.

Option	Description
Start without copyright	Starts IF/Prolog without a copyright information.
Start with consult flag set to 'nonotify'	Starts IF/Prolog with consult flag set to nonotify. No messages are displayed on the screen when a file is loaded.
Start with 'warnings' flag set to 'off'	Starts IF/Prolog with warnings flag set to off. When consult/1 is called no warnings will be displayed on the screen.
Start with package information	Shows current IF/Prolog package configuration.
Current startup configuration	Displays the current IF/Prolog configuration. IF/Prolog will be started with this configuration next time. This configuration contains also parameters set in the Configuration property page. Configuration is saved into the Registry. Configuration parameters are appended at the end of the parameters set by the user at the command line. You can disable all this parameters by adding the parameter '- -' at the end of the command line parameters. You can copy current configuration from this edit box and paste it to a file or command line. **Note:** Configuration from the Registry is used only for the GUI.
Console font	Invokes font dialog. You can modify console font.
Y Console size	You can change Y size of the console window (number of lines).
X Console size	You can change X size of the console window (number of columns).
Scrolling area	You can change scrolling size of the console window (total number of lines that are used for buffered screen output).

11.7.2 Prolog Flags Options

This is the Prolog Flags property page:

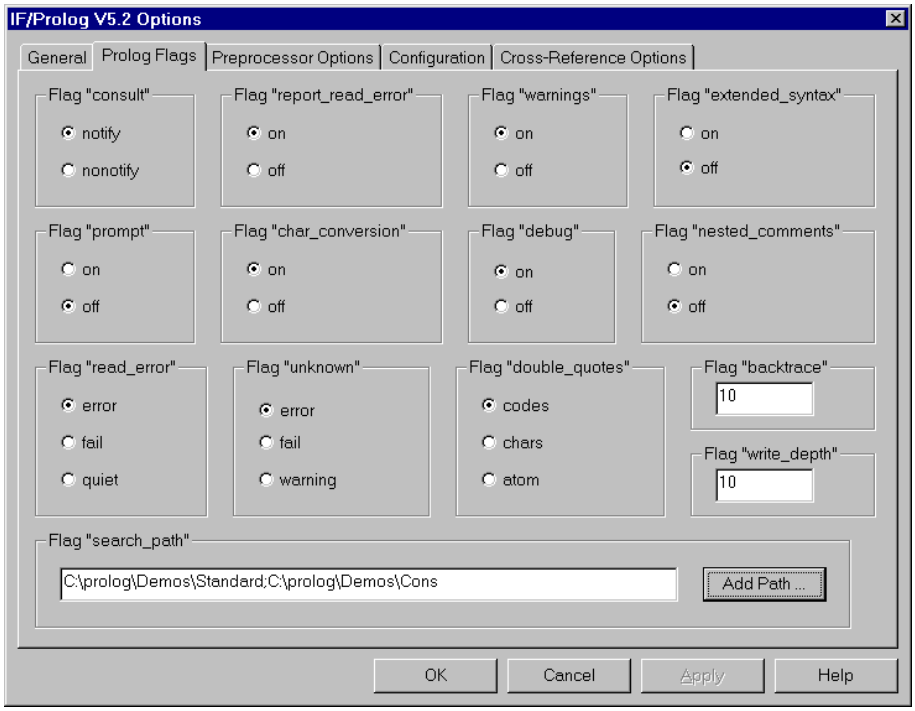


Figure: Prolog Flags property page

Prolog flags are used to make global settings for IF/Prolog. Property page contains dialog box items (radio buttons, edit boxes, buttons) by which Prolog flags can be set easily. The following table contains the available Prolog flags and their permitted values.

The following Prolog flags may be queried and modified:

Flag	Value	Description
`consult`	**notify**	A message is output when a Prolog text is loaded.
	`nonotify`	No message is output when a Prolog text is loaded.
`search_path`	*Paths*	If input files are not specified using a complete path name, they are searched for in the directories specified in the *Paths* list. The default value of this Prolog flag is the value of the system parameter `-sp`, if specified, otherwise it is the value of the environment variable `PROPATH` or it is empty.

`read_error`	**error**	Syntax errors found by analysis with `read_term/2/3`, etc. raise an exception.
	`fail`	Syntax errors found by analysis with `read_term/2/3`, etc. are reported directly by the parser in a message and the predicate fails.
	`quiet`	Syntax errors found by analysis with `read_term/2/3`, etc. are not reported and the predicate fails.
`report_read_error`		
	on	Syntax errors found by analysis with `parse_atom/6` are reported directly by the parser in a message.
	`off`	Syntax errors found by analysis with `parse_atom/6` are not reported by the parser but can be queried with the `read_error/2/3` predicates.
`warnings`	**on**	For `consult/1`, etc. warnings are output for possible errors.
	`off`	For `consult/1`, etc. no warnings are output.
`unknown`	**error**	If a predicate is to be activated which is not defined, an exception is raised. This does not apply to predicates for which the exception mode has been set to `on` (see `predicate_mode/3`).
	`fail`	If a predicate is to be activated which is not defined, backtracking is initiated, i.e. the predicate is evaluated with `fail`.
	`warning`	If a predicate is to be activated which is not defined, a warning is output and backtracking is initiated, i.e. this predicate is evaluated with `fail`. The warning is not output for predicates for which the exception mode has been set to `on` (see `predicate_mode/3`).
`debug`	`on`	For `consult/1` and `reconsult/1`, *all* predicates are implicitly declared as dynamic (see `dynamic/1`) to support program testing.
	off	For `consult/1` and `reconsult/1`, all the predicates which were not explicitly declared with `dynamic/1` are declared as static. These cannot then be modified, tested or displayed.

`prompt`	`on`	Output of the prompt in the Prolog input loop and interaction for backtracking for goals which have been successfully executed in the Prolog input loop and for which variables have been instantiated *always* take place, i.e. even if the standard input and the standard output are not terminals.
	off	Output of the prompt in the Prolog input loop and interaction for backtracking for goals which have been successfully executed in the Prolog input loop and for which variables have been instantiated take place *only* if the standard input and the standard output are terminals.
`char_conversion`	**on**	The character conversions defined with the predicate `char_conversion/2` are performed when terms are read.
	`off`	No character conversion is performed when terms are read.
`backtrace`	**10**	The maximal length of the **backtrace**-list. The list is created when an exception has been raised and contains the calling sequence of goals that has led to the exception. The elements of the list are predicate indicators (*Functor*/*Arity*). Goals that have been removed due to tail-recursion optimization (normally the last subgoal in a clause body) are not included in the list.
`double_quotes`	**codes**	Characters inside double quotes (") are to be interpreted as a list of character codes when reading a term (e.g. with `read/1/2`).
	`chars`	Characters inside double quotes (") are to be interpreted as a list of characters when reading a term (e.g. with `read/1/2`).
	`atom`	Characters inside double quotes (") are to be interpreted as an atom when reading a term (e.g. with `read/1/2`).
`extended_syntax`	`on`	Cyclic terms can be entered using the notation *Variable* `::` *Term*. When reading such a cyclic term, *Variable* is unified with *Term*. The above notation is also used by output of cyclic terms (see `=/2`).
	off	Cyclic terms can be entered in the above notation, but *Variable* is **not** unified with *Term*.

`nested_comments`	**off**	Nested comments are not allowed in a Prolog text. Begin of a comment inside a comment is ignored, which normally leads to a syntax error.
	`on`	Nested comments are accepted in a Prolog text.
`write_depth`	**10**	Maximal depth of terms output in the Prolog toplevel (see `break/0`). This applies to variable instantiations of a query and to exception messages. The predicate `write_formatted/2/3` takes this value in account, if a term is to be output with the format `%w`.

11.7.3 Preprocessor Options

This is the Preprocessor Options property page:

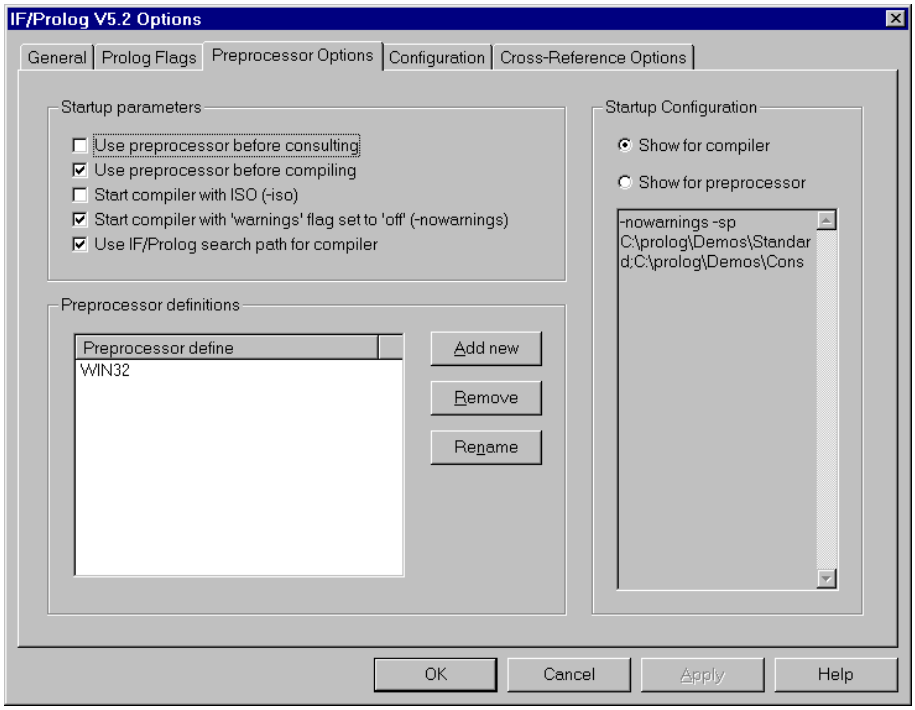


Figure: Preprocessor Options property page

Prolog files can be preprocessed before consulting or compiling. You can set this option in Preprocessor Options property page. Preprocessor is invoked only by menu or button commands. It is not invoked by predicates consult/1, reconsult/1 or compile/1.

Option	Description
Use preprocessor before consulting	Preprocessor is activated before consulting or reconsulting a file invoked by menu or button command.
Use preprocessor before compiling	Preprocessor is activated before compiling a file invoked by menu or button command.
Start compiler with ISO	Compiler is started with -iso flag.
Start compiler with 'warnings' flag set to off	Compiler is started with -nowarnings flag.
Use IF/Prolog search path for compiler	Compiler is started with -sp flag
Preprocessor definitions	List box with preprocessor definitions. Definitions (e.g. #define SYSTEM) are appended to the preprocessor parameters (e.g. -DSYSTEM) and you can use #ifdef (or #if defined(...)) blocks in your code.
Preprocessor parameters	Preprocessor parameters can be displayed as they are used for compiling or for consulting

11.7.4 Configuration Options

This is the Configuration property page:

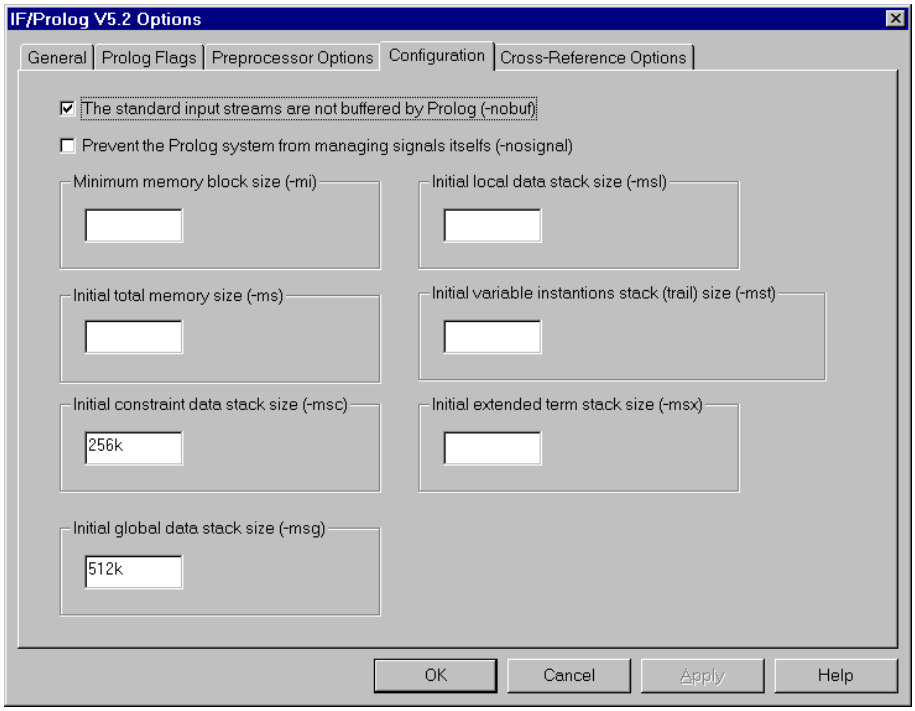


Figure: Configuration property page

IF/Prolog configuration. You can modify stack sizes, memory parameters, ...

Option	Description
-nobuf	The standard input streams are not buffered by Prolog.
-nosignal	This option prevents the Prolog system from managing signals itself, i.e. it defines no user-defined signal handling routines. The predicate signal_control/2 has no effect.
-mi	Defines minimum size of the memory blocks requested by the Prolog system. Default: 8K
-ms	Defines initial size of the total memory for the Prolog system. Default: 8K
-msc	Defines initial value of the constraint data stack. Default: 8K
-msg	Defines initial value of the global data stack. Default: 30K
-msl	Defines initial value of the local data stack. Default: 30K
-mst	Defines initial value of the variable instantions stack (trail). Default: 8K
-msx	Defines initial value of the extended term stack. Default: 8K

11.7.5 Cross-Reference Options

This is the Cross-Reference Options property page:

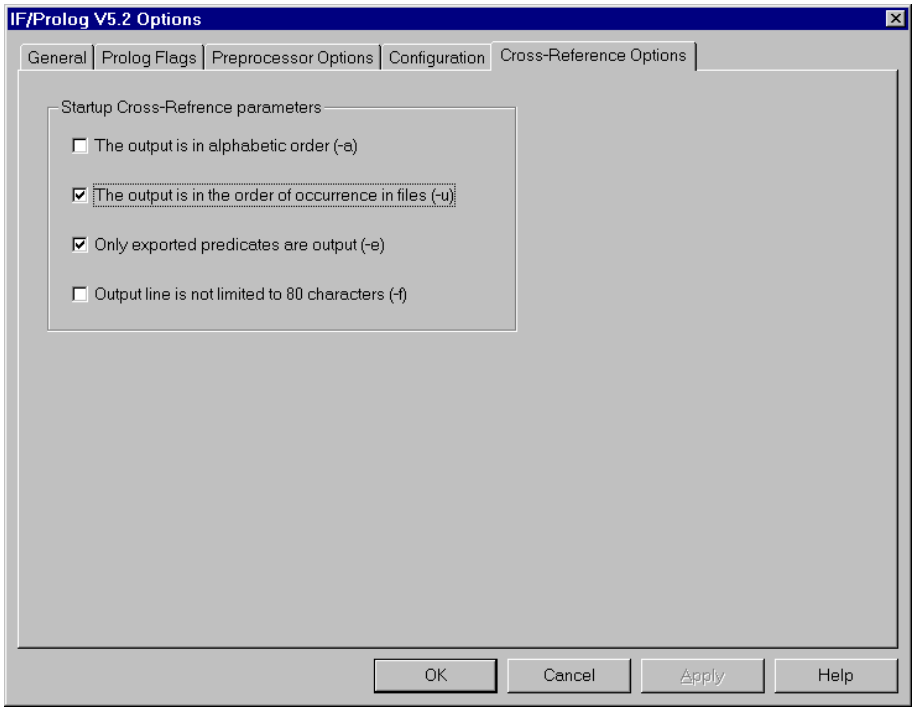


Figure: Cross-Reference Options property page

The Cross-Reference tool displays predicate information for particular file. The information includes which predicates are defined in target file, where it is implemented (in which lines), ... Options set in this property page are used only when Cross-Reference tool is invoked by menu or toolbar command. They are not used when the Cross-Reference tool is run outside the IF/Prolog development GUI.

Option	Description
-a	Cross-Reference output is in the alphabetic order - predicates are ordered alphabetically.
-u	Cross-Reference output is in the order of occurrence in files - predicates are not ordered.
-e	Cross-Reference output contains only exported predicates.
-f	Cross-Reference output is not limited to 80 characters.

11.8 Debugger Support

Graphical user interface supports debugging by several features:

- **Debugger Toolbars**
- **Accelerator Keys**
- **Breakpoint Dialog**

This is the IF/Prolog window with activated debugger:

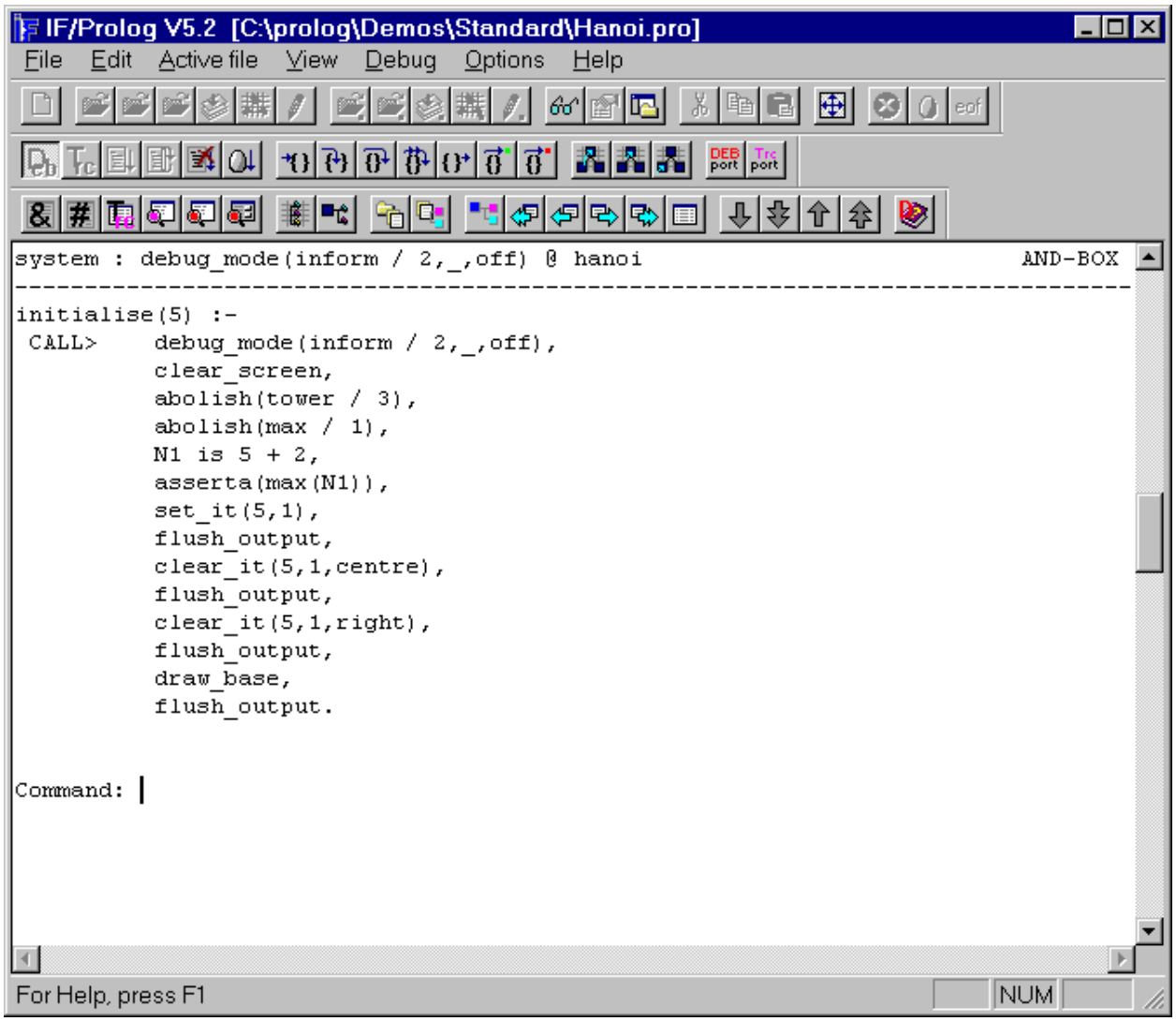


Figure: IF/Prolog window with activated debugger

11.8.1 Debugger Toolbars

Debugger toolbars contain buttons that are shortcuts for many Prolog debugger commands. It is easier to use debugger with toolbars. You do not need to remember command syntax. There are two debugger toolbars. The first is the main debugger toolbar. It contains commands for debug process control. The second toolbar contains command that provide information on debug process.

See Also
Main Debugger Toolbar, Debugger Information Toolbar

11.8.2 Accelerator Keys

Development user interface defines several key sequences as shortcuts for debugger commands.

Key Sequence	**Description**
F10	Debugger step
Shift + F10	Debugger skip
Shift + F11	Debugger fast skip
F5	Debugger continue
Shift + F5	Debugger abort

11.8.3 Breakpoint Dialog

Breakpoint dialog displays list of all current breakpoints. They can be activated, deactivated, removed or a new breakpoint can be set.

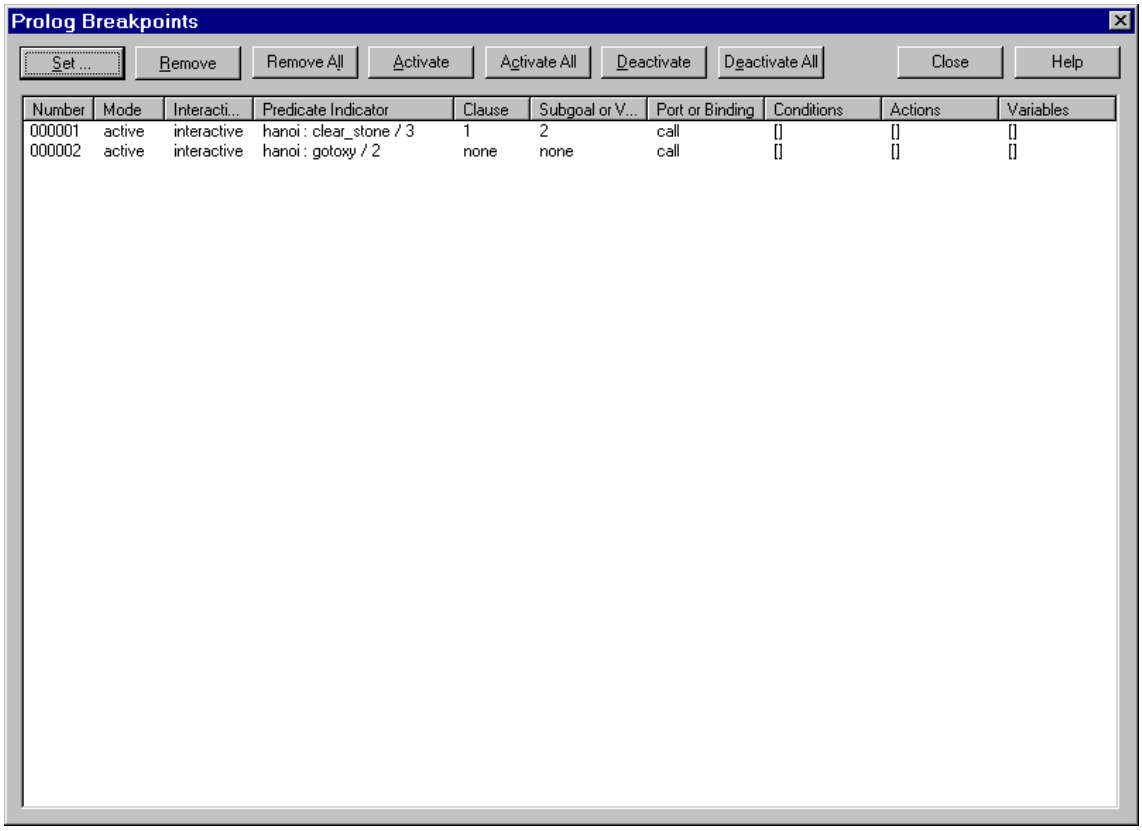


Figure: Breakpoint dialog

The breakpoint list has several fields that describe breakpoint parameters:

Field	Description
Number	Breakpoint identifier.
Mode	Mode of the breakpoint - active/suspended.
Interactivity	Interactivity of the predicate - batch/interactive.
Predicate Indicator	Predicate name, arity and its module in the format: "module : name / arity".
Clause	Clause number.
Subgoal or Variable	Subgoal (regular stop) number or monitored variable.
Port or Binding	Port (regular stop) or name of the monitored variable.
Conditions	A list of conditions on which an action is invoked.
Actions	A list of actions that are invoked on conditions.
Variables	List of variables of the breakpoint.

The table describes functionality of the dialog buttons.

Button	Description
Set	Invokes "Set Breakpoint" dialog. When there is at least one breakpoint "Set Breakpoint" dialog displays parameters of this breakpoint.
Remove	Removes current selected breakpoint.
Remove All	Removes all breakpoints.
Activate	Activates current selected breakpoint.
Activate All	Activates all breakpoints.
Deactivate	Deactivates current selected breakpoint.
Deactivate All	Deactivates all breakpoints.
Close	Closes the Breakpoint dialog.
Help	Invokes this help page.

11.8.4 Set Breakpoint Dialog

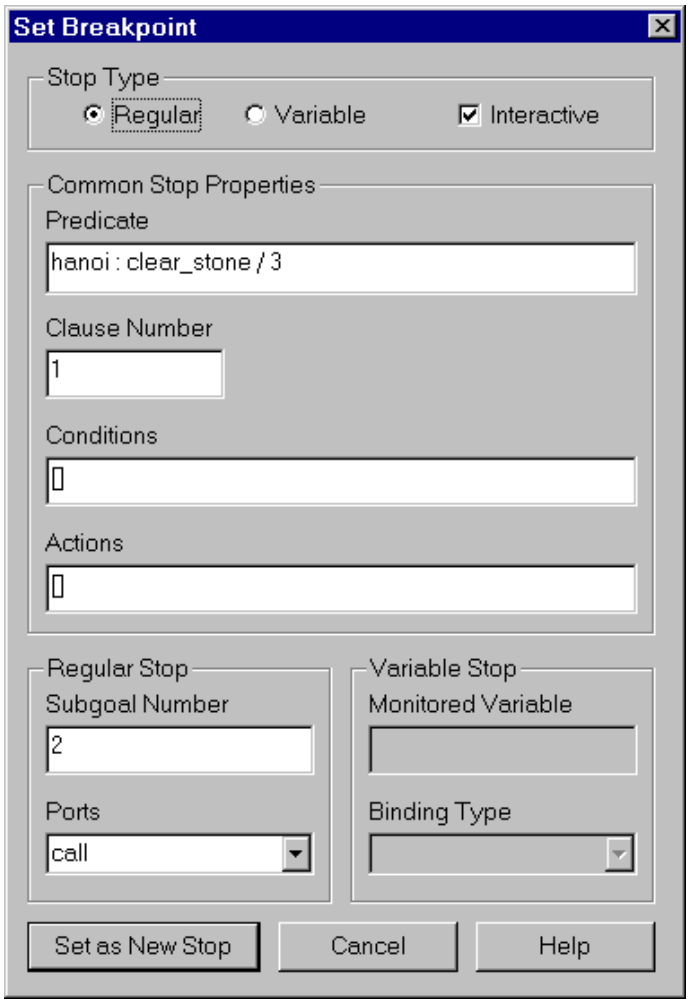


Figure: Set Breakpoint dialog

Set Breakpoint dialog allows to set a new breakpoint. This dialog can be invoked from the Breakpoint dialog. When Breakpoint dialog contains at least on breakpoint Set Breakpoint dialog fields are initialized with parameters of the selected breakpoint. You can modify it and set similar breakpoint or delete all fields and create new one.

The table describes fields and buttons of the dialog.

Field or Button	**Description**
Regular/Variable radio buttons	"Regular" sets a regular breakpoint. To minitor a variable set "Variable" breakpoint type.
Interactivity check box	Sets interactivity of the breakpoint.
Predicate field	Predicate name and arity or also its module in the format: "name / arity" or "module : name / arity".
Clause Number field	Number of clause to set breakpoint for.
Conditions field	A list of conditions on which an action is invoked.
Actions field	A list of actions that are invoked on conditions.
Subgoal field	Number of subgoal to set breakpoint for. Only for regular breakpoints.
Ports field	Port to set breakpoint for. Only for regular breakpoints.
Monitored Variable field	Name of the monitored variable to set breakpoint for. Only for variable breakpoints.
Binding Type field	Binding type to set breakpoint for (bind/modify). Only for variable breakpoints.
Set as New Stop button	Creates a new breakpoint with parameters specified in the fields of this dialog.
Cancel	Cancels the dialog and no breakpoint is set.
Help	Displays this help page.

See Also
Debugger Commands, Debugger Toolbar, Debugger Information Toolbar, Console Window

11.9 Linking Custom IF/Prolog Applications

IF/Prolog custom applications can be linked also with GUI. There are two versions of GUI:

- **Runtime User Interface**
- **Development User Interface**

Each GUI version consists of several files. They include main application loop, resources (menus, toolbars), main console window, ... To create a custom linked application you do not need to call InitProlog to iniitialize Prolog. You only define your C-predicates with your code and declare them in Cboot() function. You can define also Cshutdown() function to call custom functions during the Prolog shutdown.

11.9.1 Runtime User Interface

This version of the graphical user interface is intended to be used for end - user IF/Prolog applications. It includes simple console window, simple toolbar and menu.

This is the Runtime User Interface main window:

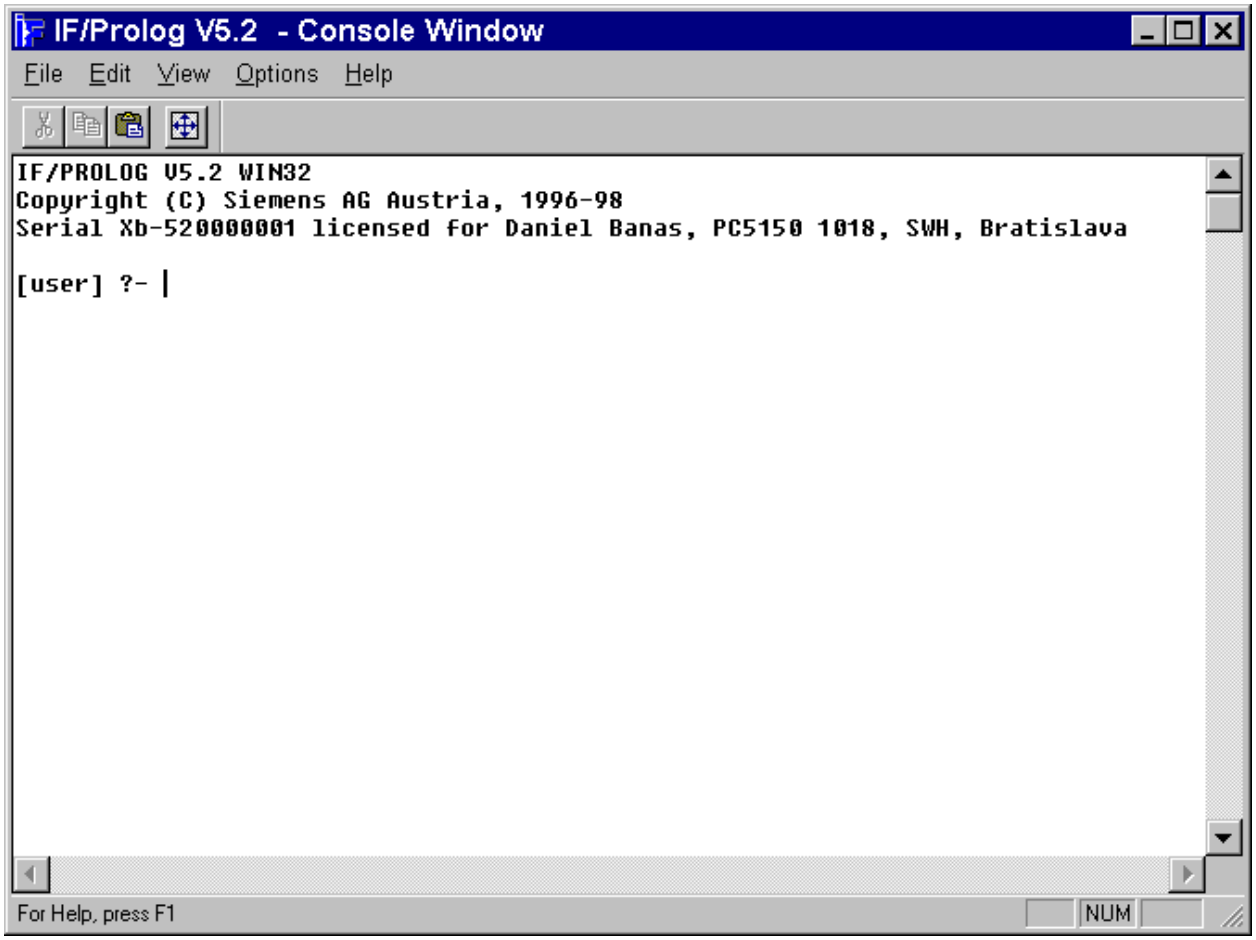

IF/Prolog V5.2 - Console Window
File Edit View Options Help
IF/PROLOG V5.2 WIN32
Copyright (C) Siemens AG Austria, 1996-98
Serial Xb-520000001 licensed for Daniel Banas, PC5150 1018, SWH, Bratislava
[user] ?-
For Help, press F1
NUM

Figure: Runtime User Interface main window

11.9.2 Development User Interface

Development version of the graphical user interface is intended to be used for debugging custom IF/Prolog applications. It includes the same interface as the IF/Prolog application.

This is the Development User Interface main window:

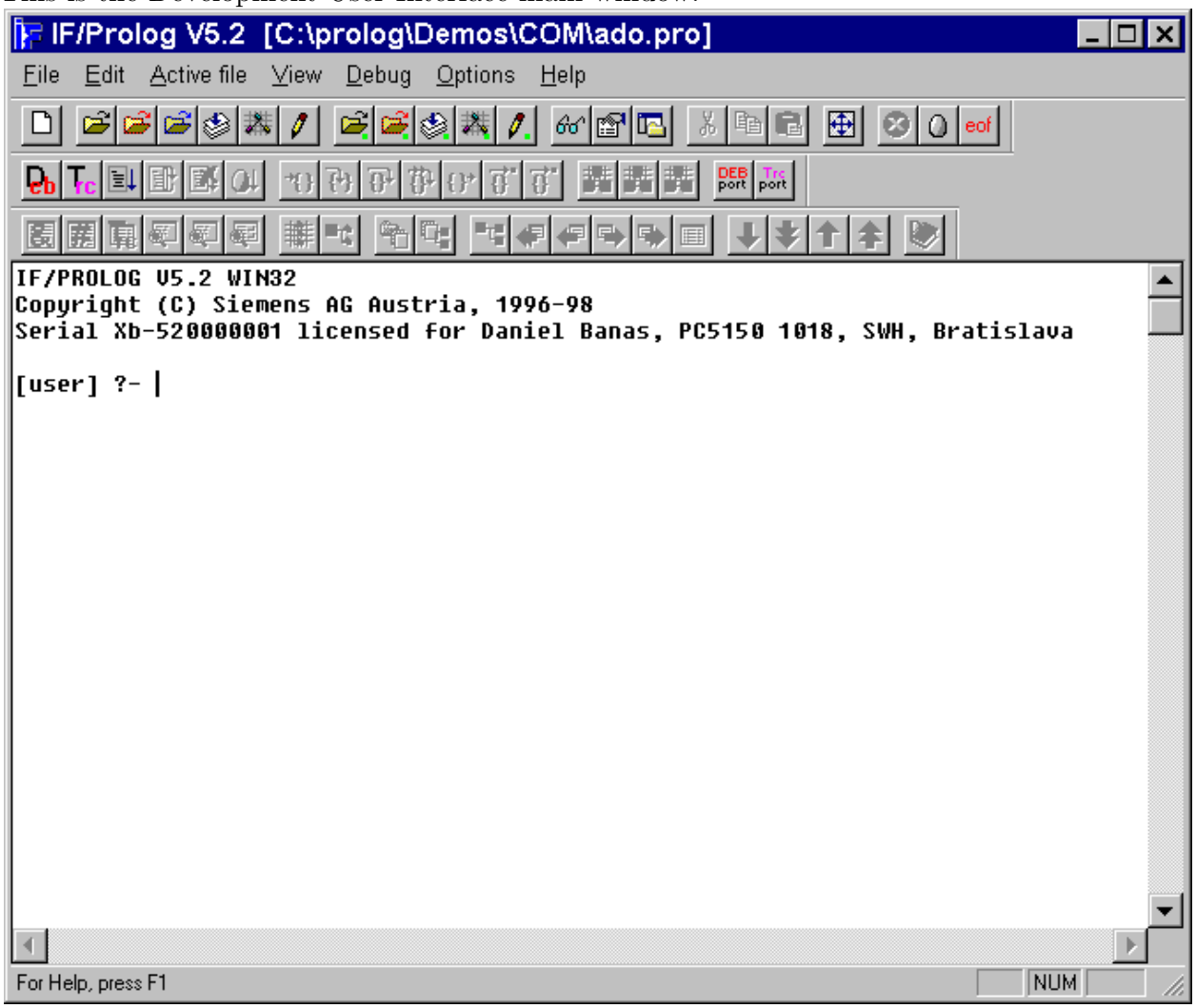

IF/Prolog V5.2 [C:\prolog\Demos\COM\ado.pro]
File Edit Active file View Debug Options Help
IF/PROLOG V5.2 WIN32
Copyright (C) Siemens AG Austria, 1996-98
Serial Xb-520000001 licensed for Daniel Banas, PC5150 1018, SWH, Bratislava
[user] ?-
For Help, press F1
NUM

Figure: Development User Interface main window

11.9.3 Release and Debug configurations

User interface libraries come with two versions - Release and Debug. You can use Debug configuration to debug the application code. The Release configuration does not contain debug information and is optimized - the resulting code is faster and smaller. Release configuration can be linked with runtime version of IF/Prolog GUI where main IF/Prolog window is simpler to create end - user version of the custom application. Debug configuration can be linked with development version of IF/Prolog GUI where main IF/Prolog window contains more toolbar buttons and tools that help the development to create test version of the application.

The table describes which files must be linked with which GUI version.

Files	**Runtime GUI**	**Development GUI**
Resource DLL for Release configuration	GUIres.dll	GUIres.dll
Resource DLL for Debug configuration	GUIresd.dll	GUIresd.dll
User Interface static library for Release configuration	rtmGUI.lib	devGUI.lib
User Interface static library for Debug configuration	rtmGUId.lib	devGUId.lib
Main object file for Release configuration	rtmGUImain.obj	devGUImain.obj
Main object file for Debug configuration	rtmGUImaind.obj	devGUImaind.obj

`Important note:` All IF/Prolog custom GUI applications have to be delivered with an appropriate MFC dynamicaly linked library - with file mfc42.dll (that is delivered with IF/Prolog package and installed in the system).

`VC++ 6.0 note:` Due to changes of MFC libraries in Visual C++ 6.0 different libraries for IF/Prolog GUI application must be linked when using Visual C++ 6.0. VC++ 6.0 compliant libraries contain the number '6' in the filenames e.g. the `devGUI6d.lib` is a user interface static library for Debug configuration containing Development GUI. Compiling custom GUI applications with VC++ 6.0 is supported from the version 5.2a Release 2.01. Custom applications copmiled with Visual C++ 6.0 have to be delivered with an appropriate versiob of the resource DLL - GUIres6.dll. IF/Prolog Application Wizard is can create Visual C++ 6.0 compatible projects from the version 5.2a Release 2.01.

This is the list of all IF/Prolog C Interface VC++ 6.0 specific files:

- Resource DLL: `GUIres6.dll GUIres6d.dll`
- Runtime GUI: `rtmGUI6.lib rtmGUI6d.lib rtmGUImain6.obj rtmGUImain6d.obj`
- Development GUI: `devGUI6.lib devGUI6d.lib devGUImain6.obj devGUImain6d.obj`

11.9.4 Special GUI startup parameters

These startup parameters are new to IF/Prolog V5.2. They are included in Windows port only and apply to GUI IF/Prolog applications only.

- **-wshow** Causes the main IF/Prolog is showed (e.g. when it was minimalised) on every write operation into this window. The default behavior is that the IF/Prolog main window is showed when waiting for the keyboard input.

- **-wmin** Minimizes main window on start-up.
- **-nobanner** Prevents from showing banner window on start-up.

GUI startup parameters apply to both GUIs - to Runtime and Development GUI.

11.9.5 Demos

You can get more information from demos. These demos are linked with GUI libraries:

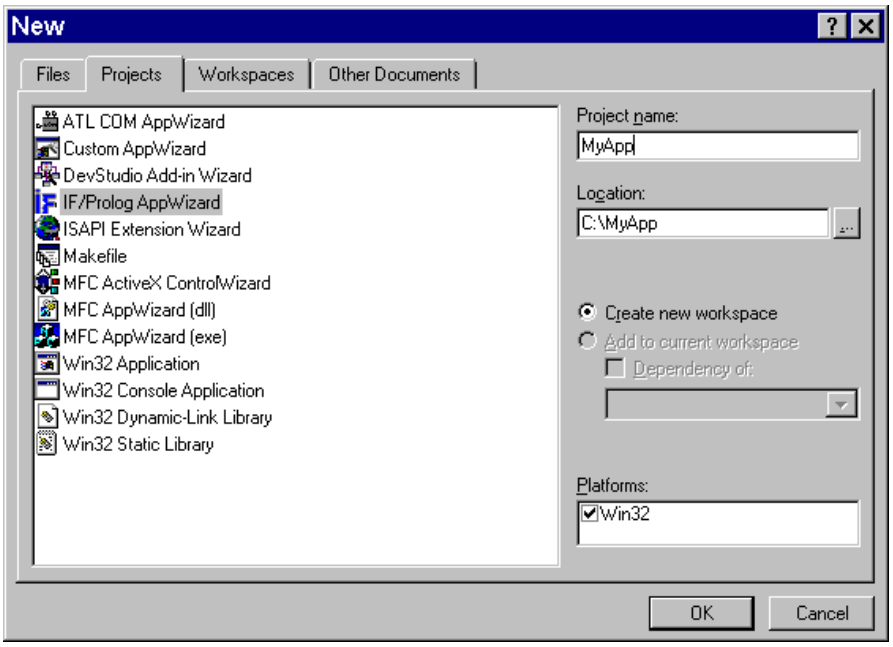
- **Visual C++\WinApp**
- **Visual C++\WinPred**
- **Visual C++\EventDemo**
- **Visual C++\Queens**

11.10 Using the IF/Prolog Application Wizard

IF/Prolog Application Wizard can create a skeleton and Visual C++ project for several IF/Prolog application types:

- Console Application - Prolog sources only
- Console Application - C/C++ and Prolog sources
- GUI Application - Prolog sources only
- GUI Application - C/C++ and Prolog sources, dialog based
- GUI Application - C/C++ and Prolog sources, MFC window based

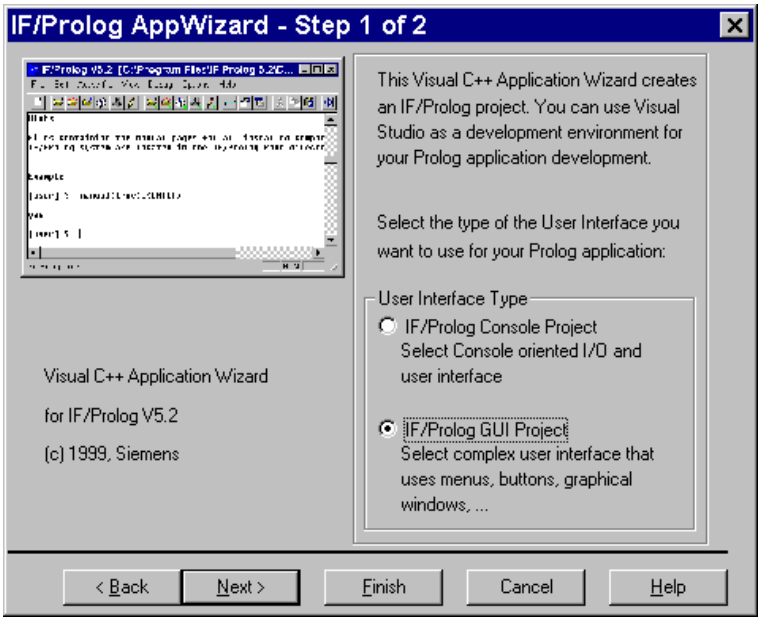
You can choose the IF/Prolog Application Wizard as one of the Visual C++ project Wizards:



The selection process includes two steps:

- `Step 1`
  Choosing the user interface type (Console or GUI)
- `Step 2`
  Choosing the project source type (Prolog, Prolog & C/C++, ...)

Step 1

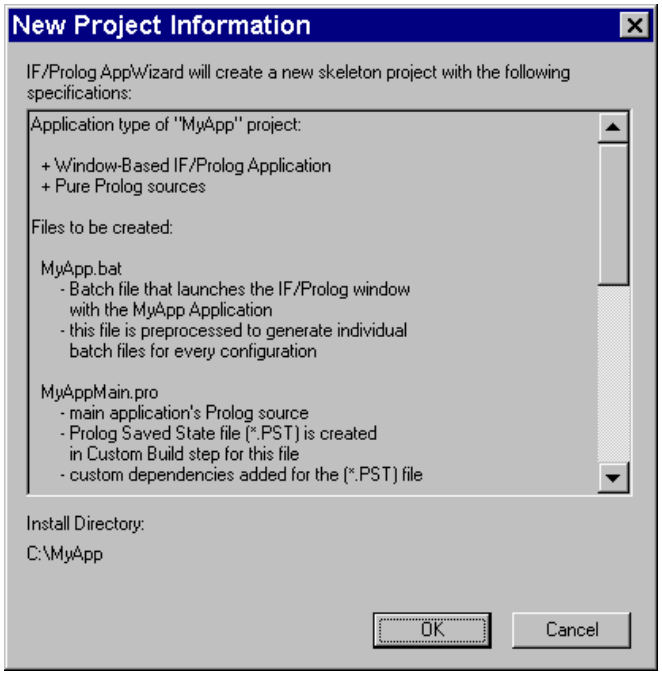

IF/Prolog AppWizard - Step 1 of 2
This Visual C++ Application Wizard creates an IF/Prolog project. You can use Visual Studio as a development environment for your Prolog application development.
Select the type of the User Interface you want to use for your Prolog application:
User Interface Type
IF/Prolog Console Project
Select Console oriented I/O and user interface
IF/Prolog GUI Project
Select complex user interface that uses menus, buttons, graphical windows, ...
Visual C++ Application Wizard
for IF/Prolog V5.2
(c) 1999, Siemens
< Back
Next >
Finish
Cancel
Help

You can select your desired user interface type - Console interface or GUI interface.

Step 2

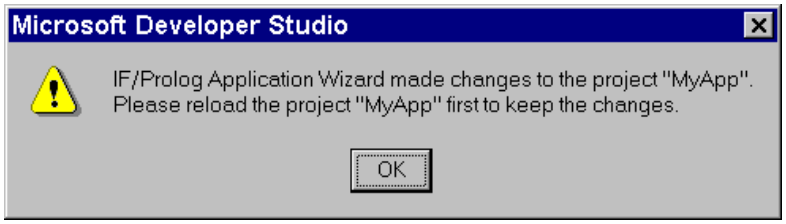
Then you can select application subtype - if your desired Visual C++ will include Prolog sources only or C/C++ and Prolog sources. GUI application type offers also Dialog based or MFC window based interfaces.

Confirmation


New Project Information
IF/Prolog AppWizard will create a new skeleton project with the following specifications:
Application type of "MyApp" project:
+ Window-Based IF/Prolog Application
+ Pure Prolog sources
Files to be created:
MyApp.bat
- Batch file that launches the IF/Prolog window with the MyApp Application
- this file is preprocessed to generate individual batch files for every configuration
MyAppMain.pro
- main application's Prolog source
- Prolog Saved State file (*.PST) is created in Custom Build step for this file
- custom dependencies added for the (*.PST) file
Install Directory:
C:\MyApp
OK
Cancel

After the two-step selection the Application Wizard will show a description of the project created. You can see what files that will be generated with a short note. After the confirmation the whole project is generated and a read-me file for this project is opened.

Special


Microsoft Developer Studio
IF/Prolog Application Wizard made changes to the project "MyApp".
Please reload the project "MyApp" first to keep the changes.
OK

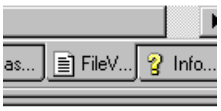
IF/Prolog Application Wizard makes some changes to the project that are not automatically loaded into Visual C++. You must reload the project by hand. This process is described in the chapter How to finish IF/Prolog project creation.

11.10.1 How to finish IF/Prolog project creation

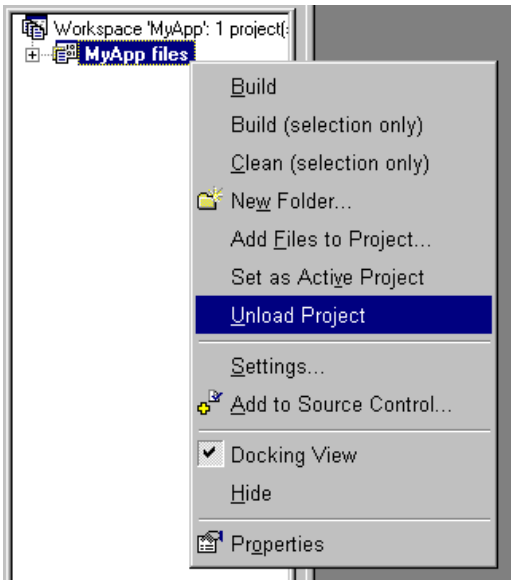
IF/Prolog project created by Application Wizard must be reloaded in Visual C++ environment to include all changes made by IF/Prolog Application Wizard.

To reload the project you have to:

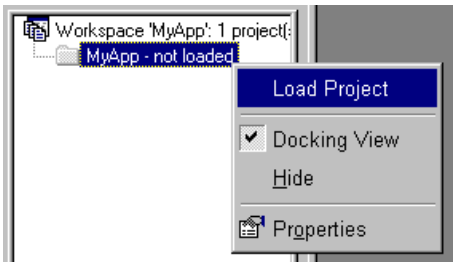
1. Open the FileView in workspace window



2. Select "Your_project_name files" item
3. Press right mouse button (on right-handed mouse)
4. Select "Unload Project" in pop-up menu



5. Select "Your_project_name - not loaded" item
6. Press right mouse button again
7. Select "Load Project" in pop-up menu



Your project is now reloaded.

11.10.2 How to create IF/Prolog Console Application (Prolog sources only)

1. Start IF/Prolog Application Wizard selecting it as a new project type in Visual C++.
2. In Step 1 select "IF/Prolog Console Project".
3. In Step 2 select "Pure Prolog source".
4. Select "Finish" button.
5. In "New Project Information" dialog box select OK button.

11.10.3 How to create IF/Prolog Console Application (Prolog and C/C++ sources)

1. Start IF/Prolog Application Wizard selecting it as a new project type in Visual C++.
2. In Step 1 select "IF/Prolog Console Project".
3. In Step 2 select "Prolog & C/C++ source".
4. Select "Finish" button.
5. In "New Project Information" dialog box select OK button.

11.10.4 How to create IF/Prolog GUI Application (Prolog sources only)

1. Start IF/Prolog Application Wizard selecting it as a new project type in Visual C++.
2. In Step 1 select "IF/Prolog GUI Project".
3. In Step 2 select "Pure Prolog source".
4. Select "Finish" button.
5. In "New Project Information" dialog box select OK button.

11.10.5 How to create IF/Prolog GUI Application (Prolog and C/C++ sources, dialog based)

1. Start IF/Prolog Application Wizard selecting it as a new project type in Visual C++.
2. In Step 1 select "IF/Prolog GUI Project".
3. In Step 2 select "Prolog & C/C++ source", "Custom dialog based GUI".
4. Select "Finish" button.
5. In "New Project Information" dialog box select OK button.

11.10.6 How to create IF/Prolog GUI Application (Prolog and C/C++ sources, MFC window based)

1. Start IF/Prolog Application Wizard selecting it as a new project type in Visual C++.
2. In Step 1 select "IF/Prolog GUI Project".

3. In Step 2 select "Prolog & C/C++ source", "Custom window based GUI".
4. Select "Finish" button.
5. In "New Project Information" dialog box select OK button.

Appendix A

Preparing Prolog applications for marketing

An application which you deliver to a customer is referred to in the following as a **Prolog application**. Such Prolog applications comprise two parts:

- the Prolog kernel system
- the runtime environment

To ensure that a Prolog application is executable at your customer's site, the customer must have purchased and installed the Prolog runtime system (license information).

Prolog kernel system

You generate a new Prolog kernel system by means of the makefile *model_make*. If you have used one of the C interfaces (active, passive or device driver) in the application, or if your customer is using further system variants such as OSF/Motif or Informix, you must enter this in *model_make* before you generate a new kernel system.

The new Prolog system is part of the application and is delivered to your customer.

Runtime environment

The runtime environment is that part of your application which is written in Prolog. You make this execution environment available to your customer in the form of a saved system state with the predicate `save_runtime/1`.

Those parts of IF/Prolog system which are written in Prolog are also available in the form of a saved system state. The runtime version of IF/Prolog is located in the file *runtime.pst* in the directory *$PROROOT/prolib*.

You must create such a runtime environment for the Prolog application that it contains all the Prolog programs and the Prolog runtime system. This has the following advantages:

- Starting the Prolog system with the application is very fast.
- The file containing the saved system state is largely unreadable.
- You deliver only a limited number of files.

Example

The following example is an illustration of how to create a Prolog application. A prerequisite here is the part written in Prolog of the application *app.pro*, which is stored in the module `app`.

```
:- module app.
:- begin_module app.

app :-
       write('I am a Prolog application'),
       nl,
       halt.

:- end_module app.
```

As the next step, you must load the Prolog programs of the application with the newly generated Prolog kernel system and the supplied runtime system (*$PROROOT/prolib/runtime.pst*) and store it as a saved system state in a file. In the following program *gen.pro*, the `init` directive is used to generate a system state in the file *app.pst*. The predicate `program/0` is the entry point to the application in the module `user`.

```
:- init((save_runtime('app.pst'), halt)).

program :- app:app.
```

Now call the new Prolog kernel system and the saved system state with the Prolog runtime system in order to generate the application saved system state. With the previously mentioned files and the new Prolog kernel system named *app_prolog*, the call would look like this:

```
app_prolog -r $PROROOT/prolib/runtime.pst -c app.pro -c gen.pro
```

Using the files *app.pro* and *gen.pro* you generate the file *app.pst* which, as a saved system state, contains all the required information and can be delivered to your customers.

i You may deliver only the Prolog kernel system and the generated saved system state to your customers. You may use only *$PROROOT/prolib/runtime.pst* to generate the saved system state.

Finally, you must also develop a distribution and installation procedure which stores the Prolog kernel system, the saved system state and any further application files at a suitable location in the file tree of the target system.

The customer will usually call the application with a shell script, which could contain the following command for the above example:

```
path/app_prolog -r path/app.pst -- $*
```

where *path* is the installation path of the application on your customer's system.

Appendix B

ASCII table

Decimal	Octal	Hexadec.		Meaning	Control
0	00	00	NUL	Null, no operation	@
1	01	01	SOH	Start of Heading	A
2	02	02	STX	Start of Text	B
3	03	03	ETX	End of Text	C
4	04	04	EOT	End of Transmission, END key	D
5	05	05	ENQ	Enquiry	E
6	06	06	ACK	Acknowledge	F
7	07	07	BEL	Bell	G
8	10	08	BS	Backspace	H
9	11	09	HT	Horizontal Tabulation	I
10	12	0A	LF	Line Feed	J
11	13	0B	VT	Vertical Tabulation	K
12	14	0C	FF	Form Feed	L
13	15	0D	CR	Carriage Return	M
14	16	0E	SO	Shift Out	N
15	17	0F	SI	Shift In	O
16	20	10	DLE	Data Link Escape	P
17	21	11	DC1	Device Control 1, Continue output	Q
18	22	12	DC2	Device Control 2	R
19	23	13	DC3	Device Control 3, Stop output	S
20	24	14	DC4	Device Control 4	T
21	25	15	NAK	Negative Acknowledge	U
22	26	16	SYN	Synchronous Idle	V
23	27	17	ETB	End of Transmission Block	W
24	30	18	CAN	Cancel	X
25	31	19	EM	End of Medium	Y
26	32	1A	SUB	Substitute Character	Z
27	33	1B	ESC	Escape	
28	34	1C	FS	File Separator	
29	35	1D	GS	Group Separator	]
30	36	1E	RS	Record Separator	
31	37	1F	US	Unit Separator	-

Decimal	Octal	Hexadec.		Meaning
32	40	20	SP	SPACE
33	41	21	!	
34	42	22	”	
35	43	23	#	Number symbol
36	44	24	$	or national currency symbol
37	45	25	%	
38	46	26	&	
39	47	27	’	
40	50	28	(	
41	51	29	)	
42	52	2A	$\star$	Asterisk, multiplication sign)
43	53	2B	+	
44	54	2C	,	
45	55	2D	-	
46	56	2E	.	
47	57	2F	/	Division sign
48	60	30	0	
49	61	31	1	
50	62	32	2	
51	63	33	3	
52	64	34	4	
53	65	35	5	
54	66	36	6	
55	67	37	7	
56	70	38	8	
57	71	39	9	
58	72	3A	:	
59	73	3B	;	
60	74	3C	<	
61	75	3D	=	
62	76	3E	>	
63	77	3F	?	

Decimal	Octal	Hexadec.		Meaning
64	100	40	@	(commercial "at" or $)
65	101	41	A	
66	102	42	B	
67	103	43	C	
68	104	44	D	
69	105	45	E	
70	106	46	F	
71	107	47	G	
72	110	48	H	
73	111	49	I	
74	112	4A	J	
75	113	4B	K	
76	114	4C	L	
77	115	4D	M	
78	116	4E	N	
79	117	4F	O	
80	120	50	P	
81	121	51	Q	
82	122	52	R	
83	123	53	S	
84	124	54	T	
85	125	55	U	
86	126	56	V	
87	127	57	W	
88	130	58	X	
89	131	59	Y	
90	132	5A	Z	
91	133	5B	[	
92	134	5C	\	Backslash
93	135	5D	]	
94	136	5E	~	
95	137	5F	_	Underscore

Decimal	Octal	Hexadec.		Meaning
96	140	60	‘	
97	141	61	a	
98	142	62	b	
99	143	63	c	
100	144	64	d	
101	145	65	e	
102	146	66	f	
103	147	67	g	
104	150	68	h	
105	151	69	i	
106	152	6A	j	
107	153	6B	k	
108	154	6C	l	
109	155	6D	m	
110	156	6E	n	
111	157	6F	o	
112	160	70	p	
113	161	71	q	
114	162	72	r	
115	163	73	s	
116	164	74	t	
117	165	75	u	
118	166	76	v	
119	167	77	w	
120	170	78	x	
121	171	79	y	
122	172	7A	z	
123	173	7B	{	
124	174	7C	|	
125	175	7D	}	
126	176	7E		
127	177	7F	DEL	Delete Interrupt (Signal)

Appendix C

Glossary

alias name: Atom associated with an open stream.

argument: An object in a Prolog structure. In the term `mother(gaia, uranus)`, for example, `gaia` and `uranus` are the arguments.

arity: A non-negative integer, representing the number of arguments in a structure.

atom: A basic object incapable of further subdivision, denoted by an identifier.

backtracking: Proving method which in the event of a failed route in a blind alley returns to the last choice point in the proof tree and checks alternative paths.

body: Condition, righthand part of a rule.

box model: Model illustrating how Prolog proves a goal.

breadth-first search: Search strategy in proof trees which considers several paths simultaneously.

built-in predicate: A predicate defined by the Prolog system.

choice point: Node in the proof tree at which multiple possibilities to satisfy or refute the goal exist.

clause: Statement about objects and relations concerning objects; a fact or a rule.

compound term: see structure

conjunction (,): Combination of statements. A statement `A , B` is true when both statements `A` and then `B` are true.

constant: An atom or a number.

constraint: Restriction for the value of one or more variables.

consult: To read in, analyse and store in the database a Prolog text, executing any directives.

context: Logical environment for the entire proof tree covered by the specified goal.

cut: A built-in predicate that removes choice points.

database: Set of predicates and all the objects managed by Prolog (excluding Prolog stacks).

declaration part (of a module): Is defined by the directives which are possible in it.

depth-first search: Search strategy in proof trees which continues to pursue only one path until that path is successful or ends in a blind alley. In the latter case, for example, backtracking can be initiated.

directive: Structure having the functor `:-` and arity 1. A directive is evaluated when Prolog text is read in.

disjunction (;): ORing of statements. A statement `A ; B` is true when at least one of the statements `A` or `B` is true.

dynamic predicate: Predicate whose clauses may be be modified during execution.

error context: Context in which an error is handled.

exception: Programmable event. In the event of user errors the system branches to the part of the program defined to handle this exception. If the exception is not caught, the system issues an error message.

export: Making a predicate visible to, and thus usable by, other modules.

expression: A term consisting of numbers and compound terms with evaluable functors.

extended term: Combination of terms using operators.

fact: Fact concerning the properties and names of objects. A fact describes a clause whose body is `true`.

flag: see Prolog flag

functor: The name of a predicate or structure.

global variable: Variable whose scope is not local to a clause and whose value is not lost during backtracking.

goal: Term whose proof is attempted with depth-first searching, backtracking and unification.

ground: A ground term contains no variables.

head: Conclusion, lefthand part of a clause.

implementation part (of a module): Contains predicate and operator definitions.

import: Making a predicate from another module visible and thus usable.

infix notation: Structure notation where the operator is located between the operands, e.g. `(A+B)`.

instantiated variable: Variable which is instantiated with a constant or a compound term.

instantiation: Substitution of a variable by a non-variable.

list: A Prolog term, either the empty list `[]` or a compound term whose functor is `'.'` and which has two arguments, the second of which is a list.

literal: Atom or structure.

load: Reading compiled modules into the database.

local variable: Variable which is visible in only one clause.

logical variable: A term which can represent any term.

metapredicate: A metapredicate has an additional argument which specifies the module in whose context it is to be executed.

module: Logical collection of predicates; a module has a module interface and a module body which are interlinked by a common module name.

module body: Private portion of the module, introduced by the `begin_module` directive and containing a declaration part and an implementation part.

module interface: Public portion of the module, introduced by the `module` directive.

operator: Atom which may occur as a functor in prefix, infix or postfix notation. The notation of the operator is determined by the predicate or the directive `op/3`. An operator has an precedence and an associativity which make it possible to determine which arguments and terms of different or matching precedence can be assigned to one another.

overflow: A calculated value is greater than the storage location provided for it.

parser: Component of the Prolog system which converts character strings into Prolog terms.

postfix notation: Structure notation where the operator follows the operands, e.g. `(A,B)-`.

predicate: A relation identified by functor and arity.

prefix notation: Structure notation where the operator precedes the operands, e.g. `+(A,B)`.

predicate indicator: A compound term *Functor/Arity*.

principal functor: Outermost functor in term notation.

procedure: An empty or non-empty sequence of clauses in a database defining a predicate.

Prolog flag: System state which can be queried and in some cases set by the user.

Prolog text: A sequence of terms, representing clauses and directives.

proof: Sequence of logical conclusions which produce a statement from facts.

proof tree: The paths taken by the system during proving of a goal can be mapped onto a tree structure; both the successful paths and also the blind alleys are included in the tree.

qualification: Notation specifying the module in which it was defined.

query: Request to the Prolog system to prove one or more goals.

recursion: Relation or object which is based on itself.

resolution: The proof method of Prolog.

rule: Statement in the form `If A, then B`, term in the database in the form `B:-A`, where A is the rule body and B the rule head.

side effect: A non-logical effect of a built-in predicate, such as input, output, arithmetic evaluation or database modification.

stack: Type of memory usage; the values are pushed onto the stack and the last value stored is the first value processed again.

statement: Description of a situation which can be either exactly true or

exactly false.

static predicate: Predicate which may not be modified.

stream: Unique designation for an open input/output channel to a file, terminal or printer, for example. This designation is created by the Prolog system and can also be referenced by means of a user-defined alias name.

structure: Term consisting of a functor and a sequence of arguments. The number of arguments is the arity of the term. A structure is written in functor notation as follows: functor, followed without intervening space by an opening left parenthesis, followed by comma-separated arguments, and a closing right parenthesis. If the arity is zero, the term is also referred to as an atom and is written without parentheses.
For structures with an arity of one or two an alternative notation may be used if the functor has been defined as a prefix, infix or postfix operator.

subgoal: Term whose proof must succeed in order that the given goal may be proved.

term: One of the objects: variable, number, atom or structure.

trail stack: Memory area in the Prolog system for temporary storage of variable instantiations.

user predicate: Predicate which must be defined by the user.

unification: Method for making two terms identical through the substitution of variables.

Bibliography

[1] IF/Prolog V5.3 Reference Manual

[2] IF/Prolog V5.3 User's Guide

[3] IF/Prolog V5.3 OSF/Motif Interface

[4] IF/Prolog V5.3 Informix Interface

[5] IF/Prolog V5.3 Constraints Package

[6] IF/Prolog V5.3 Quick Reference

[7] IF/Prolog V5.3 Windows Interfaces

[8] IF/Prolog V5.3 Java Interface

[9] IF/Prolog V5.3 BDD Package

[10] X/Open CAE (Common Applications Environment) Specification. System Interfaces and Headers, Issue 4. Prentice Hall 1994.

[11] International Standard, ISO/IEC IS 13211-1. International Organization for Standardization, 1995

[12] William F. Clocksin, Chris S. Mellish: Programming in PROLOG. Standard Edition Berlin et al.: Springer 1995.

[13] Ivan Bratko: PROLOG. Programming for Artificial Intelligence Second Edition, Addison-Wesley 1990.

[14] Leon Sterling, Ehud Shapiro: The Art of PROLOG. Advanced Programming Techniques. Cambridge Massachusetts: MIT Press, 1986

Index

OSF/Motif
  import, 77
OSF/Motif user interface, 183
  debugger, 183
  error message window, 185
  operation window, 184

abort
  signal, 155
action
  context-global, 147
active
  signal handling, 158
active C interface, 195
  functions, 197
active goal, 221
  create, 223
additional information
  exception, 151
alarm
  signal, 155
alias
  current_error, 82
  current_input, 82
  current_output, 82
alias name, 79
  for stream, 80
  keyboard, 81
  screen, 81
  user_error, 81
  user_input, 81
  user_output, 81
alphanumeric user interface
  command line, 175
  constraints for subgoals, 179
  constraints for variables, 179
  debugger, 174
  field for ancestor goals, 179
  field for AND-BOX, 177
  field for breakpoints, 178
  field for database, 179
  field for old goal and clause boxes, 178
  field for OR-BOX, 178
  field for settings, 178
  help field, 177
  information fields, 176
  message lines, 176
  operating commands, 177
  output window, 176
  status indicator, 175
  trace field, 180
analyze
  term, 199
anonymous variable, 13
ARITY
  data type, 198
arity, 7, 15, 30
ASCII
  language specific charset, 62
ask questions, 14
atom, 12
attributed grammar, 110

backtracking, 15, 46, 223
begin_module
  directive, 132
box model, 38, 205
box port
  activate (default), 162
branch commands
  debugger, 173
branches
  debugger, 174
breakpoint, 170
buffering
  input stream

disable, 64
built-in
modules, 115
predicates, 87
visibility, 120
built-in predicates, 32

C functions
built-in, 197
classes, 196
complex, 196, 205
link, 238
simple, 196, 201
C interface, 32
active, 195, 196, 205
complex functions, 205
simple functions, 201
device drivers, 228
link with Prolog, 238
passive, 195, 218
data types, 219
call
define block size, 63
define character set, 62
define memory areas, 63
dynamic predicates, 62
graphical development environment, 65
without input buffering, 64
without messages, 64
without signal management, 64
call parameter, 62
-c, 65
-debug, 62
-l, 65
-language, 62
-mi, 63
-ms, 63
-msc, 63
-msg, 63
-msl, 63
-mst, 64
-msx, 64
-nobuf, 64
-nonotify, 64
-nosignal, 64
-notty, 64
-nowarnings, 64
-program, 64
-prompt, 64
-r, 66
-root, 65
-sp, 65
-st, 66
-stream, 65
-xenv, 65
call Prolog, 5, 62
call types of predicates, 32
call, with input/output redirection, 64
change
current module, 116
editor, 89
character string
device, 79
charset
language specific, 62
Chinese
language specific charset, 62
choice point, 38
classification of Prolog terms, 198
clause, 10, 13, 27
body, 28
append, 20
enter, 19
clause box
debugger, 178
clauses
delete, 19
delete several, 24
delete single, 23
display on screen, 14
enter individual, 20
enter several, 20
modify several, 24
store, 25
client, 96
command line
alphanumeric user interface, 175
comment, 12
comment line, 19
compilation, 141
advantages of, 141

at operating system level, 142
disadvantages of, 141
during a Prolog session, 142
excluding predicates, 144
file, 141
grammar rules, 104
module, 141
problems, 143
Prolog text, 141
source file, 142
undefined predicates, 144
compiled file, 90
load, 65, 144
compiled predicate
debug, 143
compiled predicates, 31
modifiability, 143
compiled Prolog code, 145
complex C functions, 32, 196, 205
example, 209, 214, 216
framework, 210
macros, 211
conjunction (AND), 28
constraint data
stack, 63
constraints
import, 78
memory area, 73
constraints for subgoals
alphanumeric user interface, 179
constraints for variables
alphanumeric user interface, 179
consult
file, 90
context, 147
actions, 147
exception handling, 150, 153
for signal handling, 158
nesting, 147
context action
context jump, 147
exception, 147
signal, 147
context jump, 148
context action, 147
explicitly initiate, 149
initiate, 148
context-free grammar, 101
context-global action, 147
catch/3, 147
current_signal/3, 147
exception/2, 147
exception_handler/3, 147
predicates, 147
signal_control/2, 147
signal_handler/3, 148
throw/1, 148
control
signal handling, 156
convert
passive goal to active goal, 223
coroutine, 206
example, 207
exit, 206
exit and remove, 206
generate and start, 206
implementation in C, 206
link to Prolog, 208
operations, 206
restart, 206
cpred.h
include file, 198
cpro.h
include file, 222
CPU time, 69
cputime, 69
create
active goal, 223
context for exception handling, 153
passive goal, 223
csize, 69
current error output, 80
current goal, 221
current module
change, 116
query, 116
current stream, 80
default, 82
query, 82
set, 82

current_error
alias, 82
current_input
alias, 82
current_output
alias, 82
cused, 69
cut, 51
grammar rules, 110
green, 52
red, 52

data object
C, 197
convert, 197
generate, 200
map, 198
Prolog, 197
data type
MODULE, 219
t_cursor, 219
t_module, 219
t_parse_context, 219
t_query, 219
data types
passive C interface, 219
database, 13, 34, 87
example, 9
extend in a dialog, 19
modify, 10
query known modules, 115
read in a file, 12
remove module, 146
size, 69
used, 69
write to file, 91
debug, 161
compiled predicate, 143
debug mode
of a predicate, 163
query, 163
switch off, 163
switch on, 163
debug/1, 169
debug_mode/3, 163
debugger
OSF/Motif user interface, 183
alphanumeric user interface, 174
branch commands, 173
clause box, 178
debug mode, 163
exit temporarily, 170
goal box, 177
operation, 168
session, 168
settings, 161, 169
declaration part
module body, 133
default
for current streams, 82
input/output stream, 81
default database, 87
define
memory block size, 63
delivery
Prolog application, 283
depth-first search, 34
derivation, 13
device, 79
character string, 79
file, 79
null, 79
pipe, 79
socket, 79
standard, 79
string, 79
terminal, 79
device driver, 79
device.h
include file, 228
dialog
enter clause, 19
extend database, 19
difference list, 104
differences
C functions and Prolog, 205
directive, 117
begin_module, 132
discontiguous, 136
dynamic, 137
end_module, 126

export, 127
import, 133
include, 118
init, 38, 145
initialization, 38
meta, 128, 138
module, 126
multifile, 136
op, 139
private, 136
reexport, 130
selective import, 134
disable
input buffering, 64
signal management, 64
disjunction (OR), 28
disjunction of clauses, 30
DISPLAY
environment variable, 75
distribution
Prolog application, 283
DNS, 98
domain_error
error type, 151
driver model, 79
dsize, 69
dused, 69
dynamic
declare predicate, 62
directive, 137
dynamic (syn.) modifiable, 143

EDITOR
environment variable, 75
editor, 10
call, 88
call during a Prolog session, 88
change, 89
errors, 89
exception, 89
end_module
directive, 126
enter
clause in a dialog, 19
environment variable
DISPLAY, 75
EDITOR, 75
PRORC, 75
PROROOT, 75
query, 75
reset, 75
set, 75
SHELL, 75
environment variables, 74
epilog, 205
error context, 201
error message window, 185
error messages, 80
error term
exception, 151
error type, 151
domain_error, 151
evaluation_error, 151
existence_error, 151
instantiation_error, 151
permission_error, 151
representation_error, 151
resource_error, 151
syntax_error, 151
system_error, 151
type_error, 151
esize, 69
EUC
language specific charset, 62
eused, 69
evaluation_error
error type, 151
exception, 150
additional information, 151
context action, 147
error term, 151
error type, 151
raise, 152
exception handling, 150
active C interface, 201, 202
create context, 153
rules, 154
exceptions during loading, 146
execute a goal, 44
execute subgoals, 43
existence_error

error type, 151
exit debugger
temporarily, 170
exit Prolog, 6
export
predicate, 115
directive, 127
metapredicate, 128
predicate, 127
extended Prolog system
generate, 202
extended term stack
size, 69
used, 69
extended terms
stack, 64
extension
memory area, 73

fact, 10, 13, 29
failing to prove a goal, 45
field for ancestor goals
alphanumeric user interface, 179
field for AND-BOX
alphanumeric user interface, 177
field for breakpoints
alphanumeric user interface, 178
field for database
alphanumeric user interface, 179
field for old goal and clause boxes
alphanumeric user interface, 178
field for OR-BOX
alphanumeric user interface, 178
field for settings
alphanumeric user interface, 178
file
compile, 141
consult, 90
device, 79
edit, 87
load, 90, 144
load into database, 141
load operation, 146
logical, 118
modify, 13
physical, 118
read into database, 90
filename, 88
flag, 70
framework with epilog, 215
framework without epilog, 214
functor, 15
functor of a predicate, 30

generate
a new Prolog system, 238
context for signal handling, 158
generation of data objects, 200
generation of Prolog objects, 200
global
control of signal handling, 156
memory area, 73
global data
stack, 63
global stack
size, 69
used, 69
global variable
module object, 120
visibility, 120
goal, 44
execute, 223
parse, 223
redo, 47
store, 224
goal box
debugger, 177
grammar
context-free, 101
grammar rules
compilation, 104
graphical development environment
start, 65
gsize, 69
gused, 69

help field
alphanumeric user interface, 177

IF-THEN operation, 28
implementation part
module body, 133
import

module, 115
predicate, 115
directive, 133
module constraints, 78
module informix, 78
module motif, 77
predicate, 133
include
directive, 118
include file
cpred.h, 198
cpro.h, 222
device.h, 228
information
derive, 13
information fields
alphanumeric user interface, 176
Informix
import, 78
init directive, 38
execute, 145
initialization directive, 38
initiate
context jump, 148, 149
initiate backtracking, 15
input stream
buffering disabled, 64
input/output
default stream, 81
generic, 79
interface, 79
redirection, 64
stream, 79, 80
top-level dialog, 79
instantiation, 35
variable, 15
instantiation_error
error type, 151
integer
largest representable, 70
smallest representable, 70
integers
value range, 69
interactive debugger, 167
interface
input/output, 79
module, 126
interfaces
of the Prolog system, 76
interpretable, 144
interprocess communication
pipe, 92
signal, 92
socket, 95
interrupt
signal, 155
invocation
automatic, of an application, 64
ISO
standard, 62

kanji
language specific charset, 62
keyboard, 79
alias name, 81
Korean
language specific charset, 62

language
specific character set, 62
language constructs, 124
language interface
active, 195
applications, 195
passive, 195
link
C functions, 238
list
process in C, 200
load
compiled Prolog file, 144
exceptions, 146
file, 90
module, 144
suppress messages, 64
without input buffering, 64
without signal management, 64
load operation, 146
internal, 145
local
memory area, 73

local data
stack, 63
local stack
size, 69
used, 69
logic update view, 42
loops, 56

macros for including complex C functions, 211
makefile
model_make, 239
map data objects, 198
maxint, 70
memory
garbage collection, 72
query usage, 72
memory area
constraints, 73
database, 73
extension, 73
global, 73
local, 73
trail, 73
memory block size
define, 63
memory management, 72
memory parameter, 73
message lines
alphanumeric user interface, 176
messages
suppress, 64
meta
directive, 128, 138
metapredicate, 121
declare, 138
export, 128
minint, 70
mode
query for a signal, 155
model_make, 239
modifiable (syn.) dynamic, 143
modifiable predicate, 137
modification
dynamic, 144
file, 13
module
acces using variables, 220
body, 117
compile, 141
constraints, 78
current, 116
declare metapredicate, 138
define operator, 139
directive, 126
directives, 118
dynamic predicate, 137
file, 118
import, 133
informix, 78
interface, 117
load, 144, 146
local predicates, 136
motif, 77
reexport predicate, 130
reexport predicates selectively, 131
remove from database, 146
selective import, 134
module argument, 121
module body, 132
declaration part, 133
implementation part, 133
operator, 139
syntax, 132
module concept, 115
module interface of different module
impor, 133
module object
global variable, 120
predicate, 120
modules
built-in, 115
directive
export, 127
global variables, 125
interface, 126
language constructs, 124
meta-argument, 121
metapredicate, 121
module argument, 121
module body, 132

objects, 119
original module, 120
predicates, 120
query known
in database, 115

network
host, 97
service, 97
NIS, 98
non-terminal symbol, 101
notational conventions, 2
null
device, 79

objects
module, 119
occurs check, 35
op
directive, 139
operating commands
alphanumeric user interface, 177
operation window
OSF/Motif user interface, 184
operator
define, 139
option, 62
call Prolog, 62
organization
Prolog system, 61
original module, 120
output
passive C interface, 221
output window
alphanumeric user interface, 176

parameter passing
between C and Prolog, 202
parse a goal, 223
passive
signal handling, 157
passive C interface, 195
application, 222
central functions, 220
cpro.h, 222
formatted output, 221
framework, 222

passive goal
convert to active goal, 223
create, 223
passive Prolog goal, 220
pathname
specify, 65
permission_error
error type, 151
phrase, 101
pipe
device, 79
signal, 155
predicate, 30
arity, 30
built-in, 32, 87
call types, 32
compiled, 31
context-global action, 147
declare as dynamic, 137
declare as metapredicate, 138
declare as modifiable, 137
declare as private, 136
dynamic through call, 62
exclude from compilation, 144
export, 127
functor, 30
mix compiled and interpreted, 142
module object, 120
monitorable, 163
name, 30
overwrite, 146
qualify, 123
qualify implicitly, 123
reexport, 130
reexport selectively, 131
selective import, 134
trace, 164
visibility, 120
predicate indicator, 120
predicates of a module
import all visible, 133
private
directive, 136
procedure execution, 37
process management, 91

production rule, 101
production rules
  syntax, 112
program
  executable, 144
program pointer, 207
program/0, 38
Prolog
  exit, 6
  terminate, 224
Prolog application
  distribution, 283
Prolog dialog, 19
Prolog kernel system, 283
Prolog objects
  generate, 200
Prolog prompt, 11
Prolog query, 6
Prolog runtime stack
  size, 63
Prolog source code, 145
Prolog source file
  consult, 65
Prolog system
  call, 62
  environment variables, 74
  generate an extended, 238
  initialize, 223
  interfaces, 76
  organization, 61
Prolog text, 90
  compile, 141
prompt
  request, 64
PRORC
  environment variable, 75
PROROOT, 61
  environment variable, 75

qualify
  predicate explicitly, 123
  predicate implicitly, 123
query, 38
  current module, 116
  current stream, 82
  modules in database, 115
  Prolog flag, 70
  signal mode, 155
query environment variable, 75
query processing, 25
quit
  signal, 155

raise
  exception, 152
read in a file during a Prolog session, 90
read in a file when calling Prolog, 90
read into the database, 12
reconsult a file, 16
recursion, 58
redo goal, 47
reexport
  all predicates, 130
  specific predicates, 131
reexport/1
  directive, 130
reexport/2
  directive, 131
remove
  module from database, 146
representation_error
  error type, 151
reset environment variable, 75
resource_error
  error type, 151
return to Prolog, 68
rule, 10, 13, 28
rules
  for exception handling, 154
runtime environment, 283

sample session, 5
screen, 80
  alias name, 81
search path
  set, 65
selective import
  directive, 134
server, 96
set
  current stream, 82
  Prolog flag, 70

set environment variable, 75
settings for debugger, 169
SHELL
  environment variable, 75
signal
  abort, 155
  alarm, 155
  context action, 147
  interrupt, 155
  pipe, 155
  query mode, 155
  quit, 155
  termination, 155
  user_1, 155
  user_2, 155
signal handling, 154
  active, 158
  global control, 156
  in special context, 158
  passive, 157
signal management
  disable, 64
signals, 154
simple C functions, 32, 196, 201
  example, 202
size
  of dynamic memory blocks, 63
  total memory, 63
SJIS
  language specific charset, 62
socket, 95
  generic input/output, 97, 99
  input/output, 97
  bind, 97
  client, 96
  connect, 97
  create, 96
  device, 79
  domain, 96
  listen, 97
  server, 96
  type, 96
specified stream, 80
stack
  for constraint data, 63
  for extended terms, 64
  for global data, 63
  for local data, 63
  size, 69
  used, 69
standard
  device, 79
  ISO, 62
standard error output, 80
standard input, 20, 79
standard output, 20, 80
standard stream, 79, 81
start and end processing
  debugger, 170
start file, 66
  example, 68
start symbol, 101
state structure, 207
status indicator
  alphanumeric user interface, 175
stream, 79, 80
  alias name, 79
  current, 80, 82
  default, 82
  explicitly specified, 80
  information, 85
  standard, 79
STRING
  data type, 198
string
  device, 79
subgoal, 25, 43
subquery, 25
suffix for file name, 11
suppress
  messages, 64
  warnings, 64
synonym
  alias name, 82
syntax
  module body, 132
  notational conventions, 2
syntax error, 13
syntax of production rules, 112
syntax_error

error type, 151
system argument, 62
system information, 69
system modules, 115
system predicate
visibility, 120
system size, 69
system status
load, 66
system time, 69
system_error
error type, 151

t_cursor
data type, 219
t_module
data type, 219
t_parse_context
data type, 219
t_query
data type, 219
tail-recursion, 58
Taiwanese
language specific charset, 62
TERM
data type, 197
term
access to components, 199
analyze, 199
classify, 198
unification, 200
terminal
device, 79
terminal symbol, 101
terminate session, 68
termination
signal, 155
test functions
active C interface, 201
time, 69
token, 101
top level, 5
top-level dialog
error messages, 80
input/output, 79
trace
stop, 167
turn on, 165
trace field
alphanumeric user interface, 180
trace listing, 165
trace/1, 165
trail
memory area, 73
size, 64
trail stack
size, 69
used, 69
tsize, 69
tused, 69
type_error
error type, 151

unification, 34
active C interface, 200
user argument, 62, 66, 67
user_1
signal, 155
user_2
signal, 155
user_error
alias name, 81
user_input
alias name, 81
user_output
alias name, 81

value range
integers, 69
variable, 12, 14
anonymous, 13
built-in
for modules, 220
global, 125
instantiation, 15
uninstantiated, 8
version number
query, 72
visibility
global variable, 120
of predicates, 120
of system predicates, 120

visibility operator, 123
visible
    predicate from different module, 134

warnings
    suppressing, 64
working with OSF/Motif, 77
working with constraints, 78
working with Informix, 78
write the database to file, 91